Cape Peninsula
University of Technology

A UNIFIED QUERY PLATFORM FOR NOSQL DATABASES USING POLYGLOT PERSISTENCE

by

HADWIN MARQUARD VALENTINE

Dissertation submitted in partial fulfilment of the requirements for the degree

Master of Information and Communication Technology

in the Faculty of Informatics and Design

at the Cape Peninsula University of Technology

Supervisor: Dr B. Kabaso

Cape Town
May 2024

CPUT copyright information
The dissertation/thesis may not be published either in part (in scholarly, scientific or technical journals), or as a whole (as a monograph), unless permission has been obtained from the University

DECLARATION

I, Hadwin Marquard Valentine, declare that the contents of this dissertation/thesis represent my own unaided work, and that the dissertation/thesis has not previously been submitted for academic examination towards any qualification. Furthermore, it represents my own opinions and not necessarily those of the Cape Peninsula University of Technology.

04 May 2024

Signed	**Date**

ABSTRACT

The advancements in Web application technologies and IoT devices has produced an enormous amount of ubiquitous data. The increasing complexity and diverse data sources poses significant challenges for organizations in extracting meaningful insights. A unified query platform has the potential to address these challenges by providing a consolidated interface for querying and analysing disparate data sources in a unformed manner. Using design science research as a methodology, this study presents a systematic approach to designing, developing and evaluating a unified query prototype. The thesis begins by articulating the research problem in the current domain, describing the adverse impacts divergent data sources have when collating information. This study proceeds to highlight the key features necessary to realise an unified query solution.

Using Design Science research as a lens to conduct the research project, a prototype was developed as middleware to evaluate its effectiveness and efficiency. A number of components were developed that ultimately enabled the prototype to interrogate data different types of NoSQL database management systems. The aforementioned components consisted of a query parser, translator and executor designated as explicit functional features. The solution also incorporated behavioural design patterns to facilitate the entire query process. A variety of experiments were conducted to evaluate the prototypes effectiveness and efficiency. The experiments were action by a group of automated participants, each test representing as subset of a particular goal. The culmination of these results indicated the feasibility of the proposed solution.

In conclusion, while the prototype enabled the researchers to empirically analyse data, the proposed solution is a byproduct of the entire research process. Moreover, the findings of this study offers practical design and architectural insights for stakeholders seeking to enhance their understanding of unified query systems. The study recognises the challenges and opportunities in unified query systems which was expressed throughout research endeavour. This forms the basis of the recommendations and guidelines proposed by this study. Thus, contributing to the advancement of knowledge within the unified query platform systems.

ACKNOWLEDGEMENTS

I wish to express my gratitude to my supervisor Dr. Boniface Kabaso for his continuous support during my Masters studies. Dr Kabaso’s guidance, understanding and patience as I navigated through my research journey was invaluable. The completion of my research journey would have been possible without his invaluable knowledge. I would also like to extend my gratitude and appreciation to Dr Corrie Uys for the insightful analysis she performed on the prototype results.

A special thank you to my wife and daughter, Imelda and Kiera, for their endless support and sacrifices they made as I embarked on my studies.

DEDICATION

This is dedicated to my parents, my mother Murial Valentine and father, Winston Valentine. Your continued support and sacrifices made this possible.

TABLE OF CONTENTS

DECLARATION ii
ABSTRACT iii
ACKNOWLEDGEMENTS iv
TABLE OF CONTENTS vi
LIST OF FIGURES xi
LIST OF TABLES xiii
GLOSSARY xvi
CHAPTER ONE : PURPOSE AND SIGNIFICANCE 1
1.1 Introduction 1
1.2 Background to the Research Problem 2
1.3 Research Problem 2
1.4 Research Aims and Objectives 3
1.5 Research Questions 3
1.6 Research Methodology 4
1.7 Thesis Significance 4
1.8 Ethics 5
1.9 Delineation 5
1.10 Thesis Outline 6
1.11 Summary 7
CHAPTER TWO : LITERATURE REVIEW 8
2.1 Introduction 8
2.2 Theoretical Context for NoSQL systems 9
2.3 NoSQL Storage Options 10
2.3.1 Key-Value 10
2.3.2 Column-Family 11
2.3.3 Document 12
2.3.4 Graph 13
2.4 Foundations of a Unified Query System 13
2.4.1. Abstract Syntax Tree 14
2.4.2. Schema Consolidation 14
2.4.3. Query Translation 15
2.4.4. Database Integration 16
2.4.5. Output Management 16
2.5 Approaches to a Unified Query Platform 16
2.5.1. Impetus and Selections for a Polyglot Persistent System 17
2.6 Design And Architecture 18
2.6.1. Frameworks and Models 19
2.6.2. Resource Description Framework 19

2.6.3. Save Our Systems 20
2.6.4. NoSQL Abstract Model 20
2.6.5. U-Schema Data Model 21
2.7 Polyglot Systems 22
2.7.1. BigDawg 22
2.7.2. Heterogeneous Middleware by Zhang et al 23
2.7.3. NoDA 24
2.7.4. Translator Query 24
2.7.5. Apache Drill 24
2.7.6. CloudMdsQL 25
2.8 Critical Evaluation of Unified Query Systems 25
2.9 Systematic Literature Review 27
2.9.1. Planning the review 27
2.9.2. Conducting the Literature review 29
2.9.3. Reporting the review 34
2.10 Summary 34
CHAPTER THREE : RESEARCH METHODOLOGY 36
3.1 Introduction 36
3.2 Research Paradigm 36
3.2.1. Ontology 37
3.2.2. Epistemology 37
3.2.3. Axiology 37
3.2.4. This Study's Philosophical Position 37
3.2.5. DSR within the Research Paradigm 37
3.3 DSR as Strategy 39
3.3.1. DSR Process Model 40
3.3.2. DSR Guidelines 42
3.4 Research Design for the Unified Query Platform 43
3.4.1. Guideline 1 : Problem Relevance 44
3.4.2. Guideline 2 : Research Rigor 44
3.4.3. Guideline 3 : Design as a Search Process 44
3.4.4. Guideline 4 : Design as an Artifact 45
3.4.5. Guideline 5 : Design Evaluation 45
3.4.6. Guideline 6 : Research Contributions 46
3.4.7. Guideline 7 : Communication 47
3.5 Summary 47
CHAPTER FOUR : UNIFIED QUERY PLATFORM DESIGN AND IMPLEMENTATION 48
4.1 Introduction 48
4.2 System Design Goals 48

4.3 System Overview ... 49
4.2.1 Conceptual Framework ... 49
4.2.2 Applied Abstraction to the Prototype ... 50
4.4 System Design ... 51
4.4.1. DR1 : Metamodel Repository ... 51
4.4.2. DR2 : Query Parser ... 52
4.4.3. DR3 : Query Translator ... 54
4.4.4. DR4 : Query Executor ... 56
4.4.5. DR5 : Metrics Logger ... 57
4.5 System Construction ... 57
4.5.1. Query Intent ... 57
4.5.2. Query Path ... 59
4.5.3. Query Generator ... 60
4.6 System Review ... 61
4.7 Summary ... 62
CHAPTER FIVE : PROTOTYPE EVALUATION AND RESULTS ... 63
5.1 Introduction ... 63
5.2 The application of DSR to the problem domain ... 63
5.3 Experimental Overview ... 63
5.3.1. Participants ... 64
5.3.2. Procedure ... 65
5.3.3. Hardware ... 65
5.3.4. Software ... 65
5.3.5. Ethical Considerations ... 66
5.4 Experimental Setup ... 66
5.4.1. Data Models ... 66
5.4.2. Data Generation ... 67
5.4.3. Data Load ... 68
5.4.4. Data Metrics ... 69
5.4.5. Automated Tests ... 70
5.5 Experimental Results ... 71
5.5.1. Syntax and Sematic Validations ... 71
5.5.2. Retrieve complete dataset ... 72
5.5.3. Retrieve dataset where a single filter was applied ... 74
5.5.4. Retrieve dataset where a multiples filters were applied ... 75
5.5.5. Apply a limit to the dataset retrieval process ... 77
5.5.6. Apply sorting to the dataset retrieval process ... 78
5.5.7. Aggregation on a datasets ... 80
5.5.8. Update existing dataset ... 82

5.5.9. Data inserts 83
5.6 Summary 84
CHAPTER SIX : FINDINGS AND DISCUSSIONS 86
6.1 Introduction 86
6.2 Research Questions 86
6.3 Results of the Prototype Evaluation 87
6.3.1. Syntax and Sematic Validations 87
6.3.2. Retrieve complete dataset 89
6.3.3. Retrieve dataset where a single filter was applied 90
6.3.4. Retrieve dataset where a multiples filters were applied 92
6.3.5. Apply a limit to the dataset retrieval process 93
6.3.6. Apply sorting to the dataset retrieval process 94
6.3.7. Aggregation on a datasets 95
6.3.8. Update existing dataset 97
6.3.9. Data inserts 98
6.4 Significance of Results 99
6.5 Context for the Prototype's Findings 106
6.6 Implications of Findings 107
6.6.1. Prototype's Ethos 108
6.6.2. Prototype Abstraction 108
6.6.3. Query Intents of the Prototype 108
6.6.4. Query Processing 109
6.6.5. Prototype Error Handling 110
6.7 Contributions to Knowledge 110
6.7.1. Contributions to Theory 110
6.7.2. Contributions to Practices 110
6.8 Limitations 110
6.9 Recommendations and Future Research 111
6.10 Summary 113
REFERENCES 114
APPENDICES 117
APPENDIX A: Redis Schema 117
APPENDIX B: Cassandra Schema 118
APPENDIX C: MongoDB Schema 119
APPENDIX D: Neo4j Schema 120
APPENDIX E: Repository Metamodel 121
APPENDIX F: Prototype Unified Query - Template 123
APPENDIX G: Lexer Configuration 124
APPENDIX H: AST Sample 126

APPENDIX I: Data Generation - Province 128
APPENDIX J: Test Cases - Syntax and Sematic Validations 129
APPENDIX K: Test Cases - Retrieve complete dataset 130
APPENDIX L: Test Cases - Retrieve dataset where a single filter was applied 131
APPENDIX M: Test Cases - Retrieve dataset where a multiples filters were applied 133
APPENDIX N: Test Cases - Apply a limit to the dataset retrieval process 135
APPENDIX O: Test Cases - Apply sorting to the dataset retrieval process 137
APPENDIX P: Test Cases - Aggregation on a datasets 139
APPENDIX Q: Test Cases - Update existing dataset 141
APPENDIX R: Test Cases - Data inserts 142
APPENDIX S: Apdex Nonparametric Correlations 143
APPENDIX T: Source Code 145

LIST OF FIGURES

Figure. 1.1: Research Framework Adopted from Hevner et al., 2004:p.5 4
Figure. 2.1: NoSQL Data Store Types 10
Figure. 2.2: Approaches to Unified Query System (Khine & Wang, 2019:p.18) 17
Figure. 2.3: SOS Architecture (Atzeni, Bugiotti & Rossi, 2012) 20
Figure. 2.4: Example of Structural Layout (Atzeni et al, 2020) 21
Figure. 2.5: U-Schema Metamodel (Candel, Ruiz & García-Molina, 2022) 21
Figure. 2.6: BigDawg Architecture (Gadepally et al., 2016) 22
Figure. 2.7: Unified SQL Query Middleware Architecture (Zhang et al., 2021) 23
Figure. 2.8: Unified SQL Query Middleware Architecture (Tan et al., 2017) 25
Figure. 2.9: Systematic Literature Review process (Xiao & Watson, 2019) 27
Figure. 3.1: Research onion 39
Figure. 3.2: Cognition in RSDP Model (Vaishnavi, Kuechler & Petter, 2019:p.59) 41
Figure. 3.3: V & V Measurement Model (Olsen, M. & Raunak, M., 2019) 45
Figure. 4.1: Unified Query Conceptual Framework 50
Figure. 4.2: Prototype’s Metamodel 52
Figure. 4.4: AST - Add 53
Figure. 4.5: AST - Modify 53
Figure. 4.6. Chain Of Responsibility Pattern : Prototype Commands 58
Figure. 4.7: Strategy Pattern : Query Generator and Executor 59
Figure. 4.8: Visitor Pattern for NoSQL Code Generators 60
Figure. 4.9: Verification & Validation Measurement model 62
Figure. 5.1: Experimental Overview 64
Figure. 6.1: PG 1 - Errors 87
Figure. 6.2: PG 1 - Apdex Scores 88
Figure. 6.3: PG 1 - Memory Allocations Figure. 6.4: PG 1 - CPU Processing Time 88
Figure. 6.5: PG 1 - Parser Failures 88
Figure. 6.6: PG 2 - Apdex Scores 89
Figure. 6.7: PG 2 - Memory Allocations Figure. 6.8: PG 2 - CPU Processing Time 90
Figure. 6.9: PG 2 - Parser, Translator and Executor Times 90
Figure. 6.10: PG 3 - Errors 91
Figure. 6.11: PG 3 - Apdex Scores 91
Figure. 6.12: PG 3 - CPU Processing Time Figure. 6.13: PG 3 - Memory Allocations 91
Figure. 6.14: PG 3 - Parser, Translator and Executor Times 92
Figure. 6.15: PG 4 - Multiple Filter Errors 92
Figure. 6.16: PG 4 - Apdex Scores 93
Figure. 6.17: PG 4 - CPU Processing Time 93
Figure. 6.19: PG 4 - Parser, Translator and Executor Times 93

Figure. 6.20: PG 5 – Apex Scores.. 94
Figure. 6.22: PG 5 - CPU Processing Times 94
Figure. 6.24: PG 6 - Apdex Scores 94
Figure. 6.25: PG 6 - Errors... 95
Figure. 6.26: PG 6 - CPU Processing Times 95
Figure. 6.28: PG 6 - Parser, Translator and Executor Times 95
Figure. 6.29: PG 7 - Apdex Scores 96
Figure. 6.30: PG 7 - Parser, Translator and Executor Times 96
Figure. 6.31: PG 7 - CPU Processing Times 97
Figure. 6.33: PG 8 - Apdex Scores 97
Figure. 6.34: PG 8 - Parser, Translator and Executor Times 97
Figure. 6.35: PG 8 - Modification Errors 98
Figure. 6.36: PG 8 - CPU Processing Times 98
Figure. 6.38: PG 8 - Apdex Scores 99
Figure. 6.40: PG 9 - Errors... 99
Figure. 6.41: PG 9 - CPU Processing Times 99

LIST OF TABLES

Table 2.1: Alignment between Research Questions and Objectives 8
Table 2.2: SLR Questions 27
Table 2.3: SLR Search strategy 28
Table 2.4: SLR Search terms 28
Table 2.5: SLR Results 30
Table 2.6: Criteria for inclusion 30
Table 2.6: Criteria for exclusion 30
Table 2.7: Quality Assessment Question and Scoring 31
Table 2.8: SLR Results 31
Table 2.9: Search terms 33
Table 3.1: Research Perspectives for DSR (Adapted from Vaishnavi, Kuechler & Petter, 2019) 38
Table 3.2: Adapted from DSRP Model Activities (Vaishnavi, Kuechler & Petter, 2019) 41
Table 3.3: Adapted DSR Guidelines (Hevner et al., 2004; Merwe, Gerber & Smuts, 2019).. 43
Table 3.4: Research Questions and Data Collection 43
Table 4.1: Research Questions and Objectives 48
Table 4.2: Artifact: Design Requirements 49
Table 4.3: Prototype versus Equivalent Native Data Stores Features 55
Table 4.4: Pseudocode : Query Intent 58
Table 4.5: Pseudocode : Query Path 59
Table 4.6: Pseudocode : Query Generator 61
Table 5.1: Research Questions and Objectives 63
Table 5.2: Notebook Specifications 65
Table 5.3: Development system 65
Table 5.4: Database system 65
Table 5.5: Student NoSQL Repositories 66
Table 5.6: Enriched Data Generation Algorithm 67
Table 5.7: Apdex Categories 69
Table 5.8: Memory Usage Categories 69
Table 5.9: Component Execution Categories 70
Table 5.10: Component Error Categories 70
Table 5.11: Summary of Test Cases 71
Table 5.12: Syntax and Sematic Validations - Scenarios 72
Table 5.13: Syntax and Sematic Validations - Apdex 72
Table 5.14: Syntax and Sematic Validations - CPU & Memory 72
Table 5.15: Syntax and Sematic Validations - Timers 72
Table 5.16: Syntax and Sematic Validations - Error Rate 72
Table 5.17: Retrieve complete dataset - Test Sample 72

Table 5.18: Retrieve complete dataset - Apdex 73
Table 5.19: Retrieve complete dataset - CPU & Memory 73
Table 5.20: Retrieve complete dataset - Timers 73
Table 5.21: Retrieve complete dataset - Error Rate 73
Table 5.22: Retrieve dataset where a single filter was applied - Test Sample 74
Table 5.23: Retrieve dataset where a single filter was applied - Apdex 74
Table 5.24: Retrieve dataset where a single filter was applied - CPU & Memory 74
Table 5.25: Retrieve dataset where a single filter was applied - Timers 75
Table 5.26: Retrieve dataset where a single filter was applied - Error Rate 75
Table 5.27: Retrieve dataset where a multiples filters were applied - Test Sample 75
Table 5.28: Retrieve dataset where a multiples filters were applied - Apdex 76
Table 5.29: Retrieve dataset where a multiples filters were applied - CPU & Memory 76
Table 5.30: Retrieve dataset where a multiples filters were applied - Timers 76
Table 5.31: Retrieve dataset where a multiples filters were applied - Error Rate 77
Table 5.32: Apply a limit to the dataset retrieval process - Test Sample 77
Table 5.33: Apply a limit to the dataset retrieval process - Apdex 77
Table 5.34: Apply a limit to the dataset retrieval process - CPU & Memory 78
Table 5.35: Apply a limit to the dataset retrieval process - Timers 78
Table 5.36: Apply a limit to the dataset retrieval process - Error Rate 78
Table 5.37: Apply sorting to the dataset retrieval process - Test Sample 78
Table 5.38: Apply sorting to the dataset retrieval process - Apdex 79
Table 5.39: Apply sorting to the dataset retrieval process - CPU & Memory 79
Table 5.40: Apply sorting to the dataset retrieval process - Timers 79
Table 5.41: Apply sorting to the dataset retrieval process - Error Rate 79
Table 5.42: Aggregation on a datasets - Test Sample 80
Table 5.43: Aggregation on a datasets - Apdex 80
Table 5.44: Aggregation on a datasets - CPU & Memory 81
Table 5.45: Aggregation on a datasets - Timers 81
Table 5.46: Aggregation on a datasets - Error Rate 81
Table 5.47: Update existing dataset - Test Sample 82
Table 5.48: Update existing dataset - Apdex 82
Table 5.49: Update existing dataset - CPU & Memory 82
Table 5.50: Update existing dataset - Timers 83
Table 5.51: Update existing dataset - Error Rate 83
Table 5.52: Data inserts - Test Sample 83
Table 5.53: Data inserts - Apdex 84
Table 5.54: Data inserts - CPU & Memory 84
Table 5.55: Data inserts - Timers 84
Table 5.56: Data inserts - Error Rate 84

Table 6.1: Primary Research Questions 86
Table 6.2: Apdex : Model Description 100
Table 6.3: Apdex : Covariates 100
Table 6.4: Apdex : Generalized Linear Model 101
Table 6.5: CPU Utilisation : Statistical Descriptives 102
Table 6.5: CPU Physical Memory : Model Description 102
Table 6.6: CPU Physical Memory : Dependant Variable and Covariates 103
Table 6.7: CPU Physical Memory : Generalized Linear Model 103
Table 6.8: CPU Private Memory : Model 104
Table 6.9: CPU Private Memory : Dependant Variable and Covariates 104
Table 6.10: CPU Private Memory : Generalized Linear Model 105
Correlation Parameters 143

GLOSSARY

Terms/Acronyms/Abbreviations	Definition/Explanation
ACID	Atomicity, Consistency, Isolation and Durability
ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
BASE	Basically Available, Soft State, Eventually Consistent
CRUD	Create, Read, Update Delete
DBMS	Database Management System
DDD	Domain-Driven Design
DSR	Design Science Research
DSRP	Design Science Research Process
ETL	Extract, Transform and Load
GaV	Global-as-View
HTTP(S)	Hypertext Transfer Protocol (Secure)
IS	Information Systems
LaV	Local-as-View
MPP	Massively Parallel Processing
NoSQL	Not only SQL
PG	Participant Group
RESTful	Representational state transfer
RO	Research Objectives
RQ	Research Question
TCP/IP	Transmission Control Protocol/Internet Protocol
V&V	Verification and Validation

CHAPTER ONE : PURPOSE AND SIGNIFICANCE

1.1 Introduction

Information systems in the modern era has shifted the mindset of organisations from application driven processes to data driven initiatives. 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 (Košmerl, Rabuzin & Šestak, 2020; Zhang et al., 2021). This has led to the creation and adoption variety of NoSQL database technologies, each with its own underlying architectural principles (Davoudian, Chen & Liu, 2018; Oussous et al., 2018).

The NoSQL philosophy essentially stems from the shortcomings of the relational database management systems. NoSQL systems are ideal for storing unstructured and semi-structured data as it does not make use of the static table row and column concept. These data models are schema-less in nature, owing to the de-normalize data it holds within the data store (Ramadhan et al., 2020; Khine & Wang, 2019). This requires data to be interpreted by the consuming application.

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” (Khine & Wang, 2019). Hence, NoSQL technologies supports a variety of querying techniques when executing data management activities. Certain NoSQL database systems utilises APIs or RESTful interfaces to interrogate data while others utilise a derivative of the familiar SQL language (Oussous et al., 2018), i.e. commonly referred to as SQL-like. This variety empowers developers to take advantage of NoSQL technological principles and apply it to satisfy their own needs or requirements.

Within the NoSQL technology stack, four fundamental data models are supported: key-value, column-orientated, document-orientated and graph models (Davoudian, Chen & Liu, 2018). The unification of data models from four the different categories of NoSQL storage mechanisms often leads to data querying complexities (Zhang et al., 2021, Koutroumanis et al., 2021). Challenges start to arise when attempting to collate heterogenous data from disparate sources since each NoSQL database system have their own respective guidelines and features (Kolonko & Müllenbach, 2020). This is impart due to the fact that no global schema exist that is able to encompass the four fundamental data model categories associated with NoSQL. Each one is tailored to its respective NoSQL database technology serving specific use cases.

Due to this kaleidoscope of these storage technologies that exists; researchers, developers, data scientist and architects have embarked on creating a singular platform of consolidating these heterogeneous data models (Khine & Wang, 2019; Kolonko & Müllenbach, 2020). A common approach is to develop middleware, known as a polyglot persistent solution. Polyglot persistent solutions refers to a system’s ability to interface with multiple database technologies.

1.2 Background to the Research Problem

As a direct result of big data technologies, organisations face the ultimate challenge; how to query structured, semi-structured and unstructured data in a uniformed manner? In the absence of a global schema for diverse data sets (Khine & Wang, 2019; Hewasinghage et al., 2021), organisations 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 (Ramadhan et al., 2020).

The past decade has seen a rise of proposed and propriety unified query solutions to bridge the heterogenous querying gap that exists. Organisation look to extract key metrics from data to support strategic business initiatives (Khan et al., 2019; Endris, 2020; Ramadhan et al., 2020). While there has been numerous success in these endeavours, the solutions tend to serve very specific uses case and not easily generalized to the wider IT audience.

Furthermore, Hewasinghage et al. (2021:p.1) states NoSQL unified query solutions becomes "more complex as the number of participating data store types grows", often omitting certain operations of specific NoSQL data stores due feature mismatches between technologies. This is due to each category of NoSQL data store, including vendor specific artifacts, implements dissimilar mechanisms and techniques for querying and processing data models (Košmerl, Rabuzin & Šestak, 2020).

1.3 Research Problem

In the absence of a global query instrument, interrogating heterogenous NoSQL storage systems presents complexities when attempting present the data as a single unified view (Atzeni et al, 2020; Candel, Ruiz & García-Molina, 2022). According to Zhang et al. (2021: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.

As consequence of the current climate, developers 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 encapsulates how key-value, column-orientated, document-orientated and graph data models may be query via a single query mechanism simultaneously.

A prototype, named NoDA developed by Koutroumanis et al. (2021), is able to interrogate the four primary categories of NoSQL data models via a single interface. However, the prototype is unable to concurrently query datasets from NoSQL's heterogenous data models as it is only able connect and interrogate to a single data model at any given time. An approach adopted by Zhang et al. (2021) on the other hand, while being able to unify data across limited categories of NoSQL data models, it does not efficiently map the abstract query to the targeted queries.

Instead it uses wildcards in the respective targeted queries to gather data thus creating additional complexity to its runtime mapping mechanism. Furthermore, the reckless use of wildcards inherently increases the risks of error-prone software thus may cause runtime exceptions during executions (Gobert, 2020). The protype developed by Ramadhan et al. (2020), demonstrates in detail how abstract queries are mapped to the targeted queries, however it's only suited to relational type data models.

The construct of a uniformed query for the four data models of NoSQL technologies is often a by-product of the research output (Khine & Wang, 2019). Therefore a lot more in depth studies are required to address the complexities of developing a unified query responsible for generating NoSQL native queries for heterogenous NoSQL data models (Hewasinghage et al., 2021; Zhang et al., 2021). An effective and effecient way to overcome this obstacle is to develop an query platform with a standardize set of features encompassing syntax, semantics and lexical paradigms. This approach enables developers to easily interface with the heterogeneous data models while abstracting the technical details of each storage mechanism.

1.4 Research Aims and Objectives

The aim of this research study is to develop an uniformed query platform encompassing data models for each category of NoSQL storage types using a polyglot persistent technique. In order to achieve the research aim, the following objectives are set:

1. To determine a prescribed set of guidelines when creating a unified query platform for each type of NoSQL database.
2. To identify optimal de facto design and architectural principles able to translate a unified construct to native NoSQL queries.
3. To design and implement a unified query construct as middleware that collectively transforms, routes and executes an abstract query on each native NoSQL data models.
4. To determine the effectiveness and efficiency of the unified query construct considering data integration, query execution, and result retrieval.

1.5 Research Questions

The study will addresses the following research question and sub-questions:

- How can a unified query platform be developed for the four primary categories of NoSQL databases using polyglot persistent technique?
    1. What essential guidelines must be applied when building a uniformed query platform?
    2. What are the de facto design and architectural principles for developing a uniformed query platform?
    3. To what extent is the uniformed query platform able to translate abstract queries to native queries for the identified NoSQL data models?
    4. How effective and efficient is the performance of the applied query processing strategies in the unified query platform?

1.6 Research Methodology

The study adopts a quantitative approach; since the impetus for this research is to generalize the findings to a broad spectrum within the unified query domain. The intent is to utilize numerical data generated to objectively concluded the utility of the unified query solution (Vaishnavi, Kuechler & Petter, 2019; Peffers et al., 2020). The research endeavour conducted for this study utilised Design Science Research (DSR). Baskerville et al. (2018) posits that within a technological context; DRS may be used to build on existing knowledge of a particular area, thus enhancing the efficacy between humans and technology.

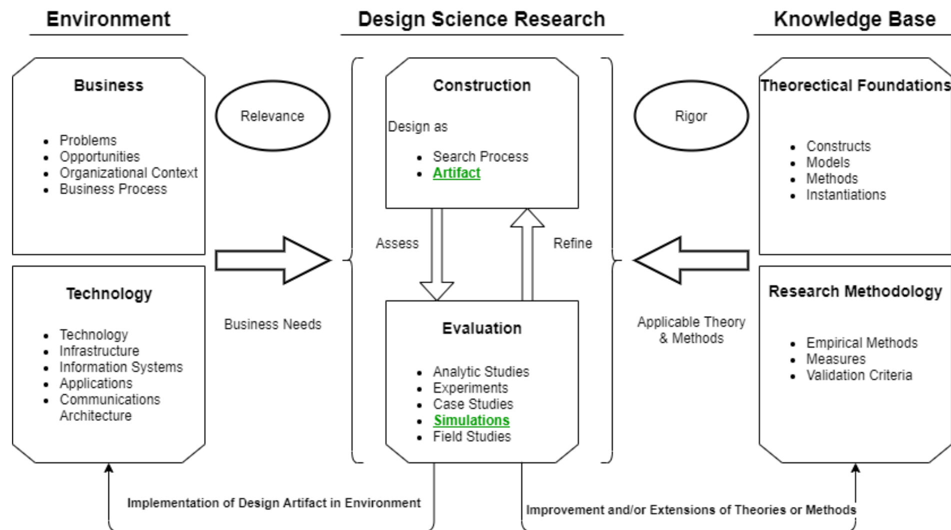


Figure. 1.1: Design Science Research Framework Adopted from Hevner et al., 2004:p.5

The theoretical framework, illustrated in Figure. 1.1, serves as the underlying structure for conducting this study. This provides the necessary research rigor distinguishing the artifact from an ordinary project endeavour. The theoretical foundations of unified query solutions will influence the decision-making and construction of the intended artifact. It is iteratively assessed and refined based on the measurable outcomes at each repetition of the research process.

Hevner et al. (2004) states that a primary motivation for using DRS is to gain understanding and new knowledge of a problem domain through a novel artifact that is able to clearly demonstrate its application. Therefore DRS was selected as the research design choice for this study, as it is most appropriate since the objectives of the study is to design, construct and evaluate an experimental prototype. The research methodology in chapter 3 articulates this to the reader as the research recipe used throughout this study.

1.7 Thesis Significance

The collation of data within organisations serves as a key factor in the strategic decision-making process. Moreover, the ability to organize the data into useful information in an optimal timeframe can be considered even more crucial. This statement is especially plausible in the current business context as data plays a huge role in an organisation's ability to react and predict market trends effectively. This study contributes to best practices and recommendations when

designing, developing and evaluating a unified query platform for the heterogenous data models of NoSQL databases. The limitations and constraints placed on the experimental prototype imposed is expressed throughout this study. The empirical data generated in a measured environment will be published detailing the input\output complexities.

The outcomes of this research endeavour provides an autonomous framework to consolidate assorted data from the four types of NoSQL storage models. The framework is essentially underpinned by exiting literature and inferences made during the research process. It reduces the need for manual intervention required from developers when interfacing disparate NoSQL databases while abstracting technical details related to specific NoSQL technology. It condenses a set of guiding design and architectural methods when developing a unified querying platform.

In summary, the significance of this study expands on existing knowledge which aids in the design and development process when creating a unified query platform, thus decreasing implementation time. More importantly, due to the principle similarities of data models within each NoSQL technological stack, this prescriptive knowledge attained may be extended to more NoSQL storage mechanisms.

1.8 Ethics

The findings of this research study will be communicated with honesty, regardless if the data indicates the unified query platform does not demonstrate its utility. Careful consideration is given to the software licensing requirements which is required to build the artifact. The software license should permit the free use and distribution of the intended software. This is important from an ethical point of view as the illegal use of software (especially propriety) may infringe or damage the reputation of all stakeholders involved. Depending on the severity, it may even lead to legality issues. The current defects of the software tools was reviewed as this may compromise the solution especially if security flaws exists compromising data.

1.9 Delineation

The scope of this study is be limited to one type of data model for each category of NoSQL databases. The research focuses developing a unified query using open source technologies which comprises of Redis (key-value store), Cassandra (column store), MongoDB (document store) and Neo4j (graph store). The actual middleware was written in C# using Visual Studio .Net Community Edition which permits usage for academic purposes without any licensing costs. Since there are a vast number of NoSQL database technologies, it will be impractical and beyond the scope of this research study to develop an all-encompassing solution for all the different types of NoSQL database technologies. This study is further bounded to specific versions of the NoSQL database implementations as any vendors changes may adversely impact the proposed solution.

1.10 Thesis Outline

This research study cover of six chapters. The following sections aims to provide an overview of what to expected within each of the chapters.

Chapter 1: Purpose and Significance

The first chapter introduces the reader to the purpose and significance of this study. It presents the reader with the research problem, aims and objectives, research question as well as a high level view on the research methodology.

Chapter 2: Literature Review

This chapter describes and analyses current literature on unified query solutions. It scrutinises existing methods, instantiations, processes and principles highlighting deficiencies and ideal practices.

Chapter 3: Research Methodology

The third chapter deliberates on the chosen research approach, strategy, data collection analysis techniques utilised in this study. This chapter proceeds to describe the research process model, guidelines and activities expressing how it pertains to this study.

Chapter 4: Unified Query Platform Design and Implementation

This chapter discusses how the experimental protype for a unified query platform was achieved by detailing the design and implementation utilized. It illustrated a conceptual framework highlighting the required components and features necessary to realise the solution. It identifies and motivates the approaches and design principles choices made to justify the final product.

Chapter 5: Protype Evaluation And Results

This segment of the research paper describes the experimental process conducted to evaluated the protypes effectiveness and efficiency. The reader is presented with documented evidence on the prototype's performance based on the empirical data collected from simulations.

Chapter 6: Findings and Discussions

The final chapter compares the protype's design implementation choices in comparison to existing solutions. It continues to meticulously detailing the findings of the research, communicating the boundaries of the prototype's performance interpreting the results attained during the testing process. It furthers acknowledges the challenges and limitations faced exploring the feasibility of the solution in regard to the problem domain. It draws insights and posits recommendations for future work on unified query platforms.

References:

The section provides a full list of cited articles, journals and books used to motivate, support and justify research themes.

Appendices:

The section further supplements research the validity of the research study.

1.11 Summary

This chapter introduces the purpose of the study and its significance. It informs the reader of the research problem and formulates an ideal resolution. The research problem is based on contemporary literature exploring the shortcomings of past solutions as a primary motivation for the research study. In addition, the chapter goes on to substantiate how DSR was used as a research method to provide the necessary rigor for this study.

CHAPTER TWO : LITERATURE REVIEW

2.1 Introduction

This chapter explores existing literature on unified query solutions. It analyses the current body of knowledge on the subject matter at hand in order to extrapolate what is known and unknown. In chapter one, section 1.4 and 1.5, the following RO's and RQ's were postulated as the impetus to describe current approaches, methods, models, instantiations and theories when developing unified query platforms.

Table 2.1: Alignment between Research Questions and Objectives

#	RQ's	RO's
1	What essential guidelines must be applied when building a uniformed query platform?	To determine a prescribed set of guidelines when creating a unified query platform for each type of NoSQL database.
2	What are the de facto design and architectural principles for developing a uniformed query platform?	To identify optimal de facto design and architectural principles able to translate a unified construct to native NoSQL queries.

This chapter firstly addresses, *RO1* motivated by *RQ1* to understand key aspects of the various types of NoSQL data stores and how to engage with each one. The study intentionally elaborated on vendor specific NoSQL database systems presented to the reader in the delineation, section 1.9. This served two purposes, firstly explicitly bring to the reader's attention the storage designation for each mentioned vendor specific NoSQL database. Secondly, the identified vendor DBMS was used in the development of the prototype. The chapter proceeds to discuss the theoretical and practical building blocks required achieve a unified query system for NoSQL databases. On this basis, the study starts with the following sections:

- Theoretical Context for NoSQL systems
- NoSQL Storage Options
- Foundations of a Unified Query System
- Approaches to a Unified Query Platform

The second objective, *RO2* inspired by *RQ2,* examined the current design and architectural practices employed by existing unified query implementations. This section aimed to expropriate and challenge the de facto standards originating in a typical unified query solution:

- Design and Architecture Practices
- Propriety, Proposed and Open-Source Solutions
- Challenges in Existing Solutions

Finally, at the end of this chapter, the literature considered the suppositions made by the relevant research works. This study made inferences based on these possibilities to conclude the ideal practices that may be packaged together to achieve the desired outcomes.

2.2 Theoretical Context for NoSQL systems

Before embarking on an implementation for a unified query solution, it is important to understand the conceptual paradigms of the BASE model and CAP theorem which directly impacts the usability and scalability of a distributed NoSQL database system. The BASE model is a concept used to manage the state of a system contrary to the ACID model. The BASE model is a less stringent approach to governing the integrity and availability of data within NoSQL databases (Khine & Wang, 2019). It prioritizes a systems performance over data consistency, hence a systems state may gradually change even if no input is processed. BASE guarantees the availability of data even under adverse circumstances:

- Basically Available - the integrity of data is not consistently guaranteed as any read operations may not provide the most recent data. Any write operations may not be persisted.
- Soft State - the state of data may change after a period of time has elapse without any immediate direct input.
- Eventually Consistent - given a set of inputs, the state of the data will eventually attain the desired or expected state.

The CAP theorem, also referred to as Brewers theorem, named after the scientist Eric Brewer who first developed it (Davoudian, Chen & Liu, 2018). The CAP theorem is a network principle affecting the data principles governing NoSQL database technologies (Glake et al., 2022). It is an extension of the BASE model, which directly influences the availability and more importantly accuracy of data interrogation techniques. This principle is rooted in the eventual consistency model comprising of the following concepts:

- Consistency - the most recent write operation will be visible to all consumers.
- Availability - ensures that data is obtainable even if there are faults with the system.
- Partitioning - system ability to operate in the event whereby a cluster may be removed or added.

This basic premise aims to describe how contending system demands are reconciled (Khine & Wang, 2019). The CAP theorem states at least one of these principles must be sacrificed as a system cannot holistically satisfy all the given states at the same time within a distributed environment (Davoudian, Chen & Liu, 2018). It is imperative to understand that for every NoSQL database that exist, it aims to satisfy this paradigm. We may reasonably draw the conclusion that it inherently affects a unified query solution depending on which aspects a NoSQL database aims to satisfy.

The theoretical concepts of governing data, as it relates to interrogation techniques, are encapsulated using mathematical formulas to justify and describe a query idea. Mathematical reasoning is a cornerstone in computer science, a rational agent motivated by empirical deduction (Endris, 2020; Roy-Hubara, Shoval & Sturm, 2022). In pursuit of developing an query construct, a variety of mathematical constructs have been applied to the diverse data models in the quest for standardization.

The idea is quite simple, if a mathematical construct is able to fully encapsulate divergent data into a single representation, this will permit the administration of probing datasets to be standardized (Oussous et al., 2018). Nevertheless, this pursuit has not been wholistically realised for heterogeneous data models. It's been well documented that relational storage mechanisms utilises relational algebra while certain solutions for non-relational data model focuses on monoid, a branch of abstract algebra rooted in a calculus to capture data aggregations (Khan et al., 2019, Hewasinghage et al., 2021; Glake et al., 2022).

Certain NoSQL database implementations are proponents of set theory since aggregations naturally occurs in these storage mediums. In graph data stores though, this mathematical model is not applicable as graph theory is the more suitable abstraction to better describe the strength of relationships between entities (Davoudian, Chen & Liu, 2018). Therefore common ground must be reached whereby the interchangeable use of applying various mathematical constructs to heterogeneous data models could be realised.

2.3 NoSQL Storage Options

The characteristics of big data technologies make NoSQL storage technologies an ideal proponent as it is able to sustain a wide range of complex data structures with data handling capabilities (Oussous et al., 2018; Candel, Ruiz & García-Molina, 2022). Hence it becomes imperative to understand how the various data types, features, the degree of complexities and relationships within the heterogeneous data models operates. By gaining a full comprehension of the data models dictates on how to engage with underlying databases. Thus providing a guideline for the development of a unified query platform. While an assortment of NoSQL databases exists, it is entrenched four distinct types of data stores; key-value, column-family, document and graph.

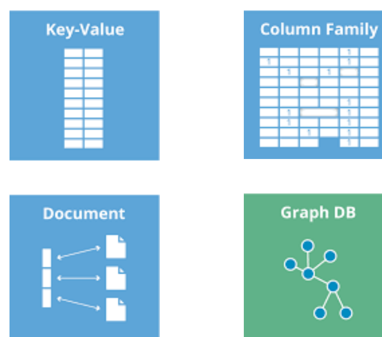


Figure. 2.1: NoSQL Data Store Types

2.3.1 Key-Value

The key-value stores are the simplest form of NoSQL database management systems (Candel, Ruiz & García-Molina, 2022). This database uses a dictionary concept whereby a unique key is served as an identifier to locate the data value or blob within a database (Oussous et al., 2018). The data value is defined as an attribute of the key, stored in an encoded or serialised format thus requiring the consuming application to decode it. Although this storage mechanism is quite

simple, consumers should be aware beforehand of the format of the data, as the model does not intuitively describe the structures.

Best practices should therefore be applied as the length of the keys directly impacts the performance and usability (Davoudian, Chen & Liu, 2018). If the length of key is too long, performance will degrade. Should it also not be in a specific format or application-defined schema, the consuming applications will be prone to errors during the translation process.

Therefore, applying a format to the data value portion of the model makes for an easier and more efficient interpretation (Khine & Wang, 2019). Due to the inherent schema-less structure within key-value data stores, indexing is not supported on data values for fast data retrieval. While this approach is suited for many use cases where fast access is not required, there is a need for indexing the values to perform atomic operations. NoSQL data stores such as Redis has addressed this issue by enhancing the key-value model by introduction data lists where complex data can be indexed and aggregation operations executed.

Redis is an example of an in-memory key-value database which stores data on computer machine's RAM (Atzeni, Bugiotti & Rossi, 2012). Unlike the other data stores, it does not have a SQL-like engine for performing data manipulations. Instead the Redis offers a list of commands to retrieved, add, update and remove data (Davoudian, Chen & Liu, 2018). Additionally, it supports functionalities such as push and range commands commonly associated with manipulating lists.

Applicable Use Cases : key-value stores are commonly used in applications to manage a user's session or personalised data that can be loaded and retrieved efficiently.

2.3.2 Column-Family

Column-family database management systems are known by a variety of names: wide-column, column-orientated and extensible record stores. Column-family data stores represents data an in multi-dimensional format consisting of columns and rows. i.e. a two-dimensional key-value or matrix store (Khine & Wang, 2019). Related of pieces of data are grouped and serialize in a single column.

The storage method enables compression as missing values does not consumed disk space as in relational storage methods (Davoudian, Chen & Liu, 2018). The clustering of related column(s) serves as the primary key enabling the manipulation of columns at the point of execution. The rows serve as the discretionary secondary key. The embedded data is dynamic in nature therefore supporting indexes on both column and rows for fast retrieval and quick data manipulation.

Google encouraged the enrichment of this data store; since then many variations of this implementation was produced. Apache Cassandra is one such technology. An open source

column-orientated database handling large dataset over many servers. The intrinsic complex nature of column-orientated data models in general requires an understanding of hashing algorithms, modelling concepts and integration details of the data store and application technology (Davoudian, Chen & Liu, 2018). This prompted the development of Cassandra's declarative query language, Cassandra Query Language (CQL), to fully realise its potential.

Cassandra Query Language (CQL) is the recommended application programming interface model for engaging with the Apache Cassandra database (Khine & Wang, 2019). It abstracts the low-level complexities via command-line or graphical interfaces. The command line interface, *cqlsh*, is distributed with each Cassandra database system which connects to a single node within the distributed network. CQL offers a rich variety data manipulation operations to engage the database such as creating key space, tables, CRUD operations, various aggregations and filtering mechanisms.

Applicable Use Cases : Column-family stores are best suited for data-intensive applications that performs data mining or analysis on large record sets. Due to its distributed clustered architecture, it's able to process big data in timeous manner.

2.3.3 Document

The document-orientated database technology can be viewed as an extension of key-value store DBMS. The fundamental difference between document and key-value data stores are that in document-orientated; the internal structure utilised describes its data content whereas with key-value data, it is not clear (Khine & Wang, 2019; Hewasinghage et al., 2021). The transparency of the data structure permits indexes to be created on attributes thus permitting efficient query capabilities. It's proficiency in interrogation techniques extends to partial data retrieval whereby consumers can specify a subset of data to be retrieved within the document. (i.e. become extremely complex with embedded structures)

The data store supports dynamic and complex structures to such as JSON, BSON or XML (Davoudian, Chen & Liu, 2018). These data formats are well supported in programming languages. Therefore the transformation layer which encapsulates data from the database to application source code enables a seamless integration. This decreases the object impedance mismatch, which relates to data not being able to be fully represented within a database due to the object-orientated programming paradigm. This is common in problem within relational schemas.

MongoDB is document database offering a flexible storage mechanism for data in a BJSON format. It closely emulates relational database system functionalities by providing compounded data to be persisted along a power SQL-like query language (Atzeni, Bugiotti & Rossi, 2012). As with many NoSQL data stores, MongoDB can be queried via multiple languages but MQL is the preferred standard as it is tailored for the database. MQL invokes a sense of familiarity since it so closely resembles the syntax and semantics of JSON, reduces the learning curve

significantly. Besides the basic CRUD operation, it offers an extensive list of grouping and filtering operators including pre-packaged functions.

Applicable Use Cases : Document data stores are especially utilised in Web 2.0 technologies such as Content Management Systems or streaming applications. These type of systems allows consumers to manage and deliver various types content in a timely manner.

2.3.4 Graph

Graph data stores came to fruition to address a need to describe relationships and dependencies between entities (Khine & Wang, 2019). This data model offers a rich variety hierarchical concepts hence making it more complex to manage. Graph data stores applies key mathematical concepts of graph theory such as nodes, edges and properties to manage and process data entities (Cox et al., 2020; Candel, Ruiz & García-Molina, 2022). The stored data is viewed as an assortment of nodes, each node contains an edge which describes the relationship between itself and other nodes. The edges are ranked indicating the strength of the connection. This bond description or metadata between nodes directly influence the dataset being queried in a single operation since the relationships are bi-directional.

Neo4j is a graph database enabling data to be represented in diverse of ways based on the relationships the existing between data. It makes use of the Cypher query language offering the basic CRUD operations for data manipulations (Diogo et al 2019). It is also worth noting that Neo4j is one of the older NoSQL database systems in existence, making it well supported and documented.

Applicable Use Cases : The application of graph data store technology is applied to a wide range of domains such as Social Networks, Medical field, AI Knowledge Graphs, Fraud Detection system, Logistics in supply chain and various other areas. The relation mappings with a graph allows for a 360 view, thus providing a complete picture of status quo.

2.4 Foundations of a Unified Query System

Authors appraising unified query systems are in agreement that there should be certain elements in place when developing solutions in order for it to be constituted as acceptable (Gadepally et al., 2016; Glake et al., 2022). This serves as a the backbone of an unified query resolutions. These guiding principles aims to simplify the heterogeneity that exists between the different data storage mechanisms (Tan et al., 2017; Ramadhan et al., 2020). It aids in directing the abstraction process necessary for addressing the complexities associated with the collection of disparate database technologies. Based on examination of the literature, this study identifies five key guiding principles:

- Abstract Syntax Tree
- Schema Consolidation
- Query Translation
- Database Integration

- Output Management

2.4.1. Abstract Syntax Tree

An Abstract Syntax Tree (AST) is used throughout computer science since it serves as an intermediatory to bridge the gap between conceptualisation, design, implementation and execution irrespective of the underlying technology used. This concept has been applied to many research areas such as source code compilers, security discovery, anti-plagiarism detection and code analysis systems (Duracik et al., 2020; Yang, Zhang & Tong, 2022). In the context of this study, AST applies to query parsers which ensures a command adheres to syntax, semantic and lexical rules. In other words, the command must be a well-formed statement (Zhang, 2020).

AST structures are representative of nodes executing logic using an if-condition-then statement to transverse through its tree structure. It characterises the structure and function of a statement via it syntactic, semantic and lexical notions. As indicative of its name, ASTs are a tree-based structure, consisting of terminal and non-terminal nodes. Non-terminal nodes provides the grammar and mechanical information while the terminal nodes defines the syntax tokens specifying a range of alphanumeric keys (Zhang et al., 2021; Yang, Zhang & Tong, 2022).

Essentially, syntax tokens are a collection of lexicons which is represent a single token or more explicitly a word in text-based parsers (Zhang, 2020). The blend of these nodes via recursion denotes compliance rules for adherence and provides lexical meaning to the intended operation (Duracik et al., 2020; Guo et al., 2020). Thus, the culmination of a desired set of nodes generated is super imposed on a query language to ensure it is well-formed. This ideology is used to build syntax and semantic rulesets instructing how a unified query may be constructed and validated.

2.4.2. Schema Consolidation

A central feature when developing unified solutions is a complete view on each underlying storage mechanism's schema information(Gobert, M., 2020). This generally known as meta-modelling. NoSQL is promoted as schema-less, due its ability to efficiently manage unstructured data. However, there is in fact a schema in place. Depending on the vendor, schema constraints may be dictated which the consuming application must adhere to.

With this in mind, the evolution of schemas are generally delegated to the application layer of a system i.e. "schema-on-read" (Candel, Ruiz & García-Molina, 2022:p.2) and not within the database engine itself. The database generally manages the basic schema elements that ought to be in place. This is exactly why NoSQL storage systems are ideal for unstructured data as the database engine does not stringently enforce any schema rules on any physical data changes when compared to relation storage system.

A more pragmatic view classifies the data structure as dynamic which changes over time as the data evolves. The curation of schemas is of upmost importance across the heterogenous NoSQL storage mechanisms, as it is needed to determine how and what data to query (Gobert, 2020). Therefore, the amalgamation of each NoSQL data model needs to be reconciled into a single abstraction often referred to as a global or federated schema (Kolonko & Müllenbach, 2020).

A re-occurring use case within a unified query system is the fact that heterogenous databases shares data for particular domain (Ramadhan et al., 2020). Inevitably, the data across the different databases for a domain will indeed have common attributes. Nevertheless, the structure of each database system will certainly be disjointed since each storage mechanism will have its specific attributes that are not shared, unique to the database instance, utilised for its own purposes (Oussous et al., 2018; Glake et al., 2022). A common use case within these federated systems is to generalize common attributes thereby forming the global schema.

The act of generalizing primarily involves to a two-step process of schema-matching and schema-mapping. Schema-matching refers to the association of an external structure to internal structures of targeted databases. Schema-mapping on the other hand describes how data is translated between external structure to internal structures. This process is quite labour intensive therefore a number of tools have been developed to efficiently manage it. Thus concluding that any changes in a schema may adversely affect the functional performance of an unified implementation. Therefore solutions tend to have a consolidated schema and maintenance aspect to mitigate potential errors.

2.4.3. Query Translation

This specific feature is arguably the most important aspect of any unified solution. It deals with the ability to generate native queries able to interrogate of NoSQL storage models (Khan et al., 2019; Koutroumanis et al., 2021). It should be noted, this feature is highly influenced by the unified approach discussed in section 2.4. Conceptually though, irrespective of the approach, it generates native queries that are able to execute on their respective NoSQL databases.

The lack of standard practices existing within this area predominantly has produced a number of naming variations often confusing readers as to what it actually entails. Zhang et al. (2021) cites this phenomenon as a computing layer. Hewasinghage et al. (2021) refers to it as a query representation or generator, whereas Ramadhan et al. (2020) and Shrestha, S., (2021) describes it as query mediator. Khan et al. (2019), on the other hand, defines this feature as a query planning component. The variation of terms used suggests how this singular unambiguous concept is applied differently in each propriety and open-source unified query solutions. It is therefore reasonable to conclude that this make strong case to further solidifying a case for standardization.

2.4.4. Database Integration

In section 1.3 of this study, the reader was introduced to the interfacing intricacies that exists when interacting with an array of NoSQL databases. For every unified query solution that exists, it inevitably has to make provision to communicate to the targeted databases (Kolonko & Müllenbach, 2020). This part of the integration is often overlooked as not enough emphasis is given to how communications with each database should be established and subsequently managed. NoSQL databases are known for implementing diverse protocols as a communication medium to access the data source (Koutroumanis et al., 2021; Zhang et al., 2021). These communication protocols ranges from HTTP(S) to TCP/IP, essentially using an adaptor or driver which implements a generic interface to connect to a database. An interesting observation made during the course of this study; there seems to be a direct correlation between primary communication protocol and query language. Depending on the protocol, the query interrogation mechanism may either access the data on the database via an API endpoint or some sort of lower level network protocol where data is streamed.

2.4.5. Output Management

In order for data from various storage systems to be presented in a uniformed manner, there are generally two approaches employed by unified query systems. This more commonly known as Global-as-View (GaV) and Local-as-View (LaV) where unification of data is achieved via it mediator (Endris, 2020; Ramadhan et al., 2020). It should be noted this also serves as input the aforementioned key features. This feature is classified as a mediator, an intelligent layer that holds structural knowledge of the local data stores. The GaV is an approach for integrating schemas of the underlying local data stores thus providing a single view of heterogenous structures. LaV on the other hand is an approach whereby local schemas are amalgamated to form a global view.

Each the approaches has its advantages and disadvantages. LaV implementations are loosely coupled therefore able to easily catalogue a new data store within its mediator (Glake et al., 2022). However, if it known to render partial results or even omitted if any new data source information is well defined in the intermediatory layer. GaV is simpler approach, static in nature providing more control in the mediation process; i.e. requires manual intervention to frame any new data points to be included.

2.5 Approaches to a Unified Query Platform

Research papers describes a number of approaches when designing and developing unified query systems (Zhang et al., 2021; Candel, Ruiz & García-Molina, 2022; Glake et al., 2022). Terms related to this research area are often used interchangeable creating further uncertainty when attempting to defined exactly which approach is being applied . Due the this subject matter being open-ended, little attention is given to exact terminology, therefore terms such as “polystores”, “multi-model”, “multi-store”, “multi-database” and “polyglot systems” are used to describe approaches to query systems supporting heterogeneous data models. Primarily there are two methods, multi-model and polyglot persistence. The multi-model approach supports

heterogeneous data models within a single database management system (Košmerl, Rabuzin & Šestak, 2020; Ramadhan et al., 2020;. Whereas, a polyglot persistent solution makes use of an abstract layer or middleware to interrogate heterogeneous databases. (Khine & Wang, 2019). This study focuses on a polyglot persistent approach for unified query systems.

2.5.1. Impetus and Selections for a Polyglot Persistent System

The motivation for the polyglot persistent approach is due to the accepted premise that the "one size fits all" approach is not suitable given the current climate of big data technologies (Khan et al., 2019:p.9598; Gadepally et al., 2016; Candel, Ruiz & García-Molina, 2022). The "one size fits all" is an approach multi-model storage mechanisms aims to satisfy. It enables support for a multitude of data models with less operational overhead when compared to a polyglot approach (Tan et al., 2017; Glake et al., 2022). However, the multi-model approach implicitly imposes limitations since any support to accommodate new data models are not easily extendable as it needs to be natively provisioned.

On the other hand, the foremost resolve for a polyglot persistent system is aimed towards flexibility and extendibility (Košmerl, Rabuzin & Šestak, 2020). This makes a polyglot persistent solution a more fluid approach as any additional data stores can be more easily accommodated and scaled. Additionally, the storage mechanisms do not share resources, making the solution completely decoupled . As Khine & Wang (2019) suggests, emerging database technologies will always evolve as the current digital climate continues to reach emerging markets.

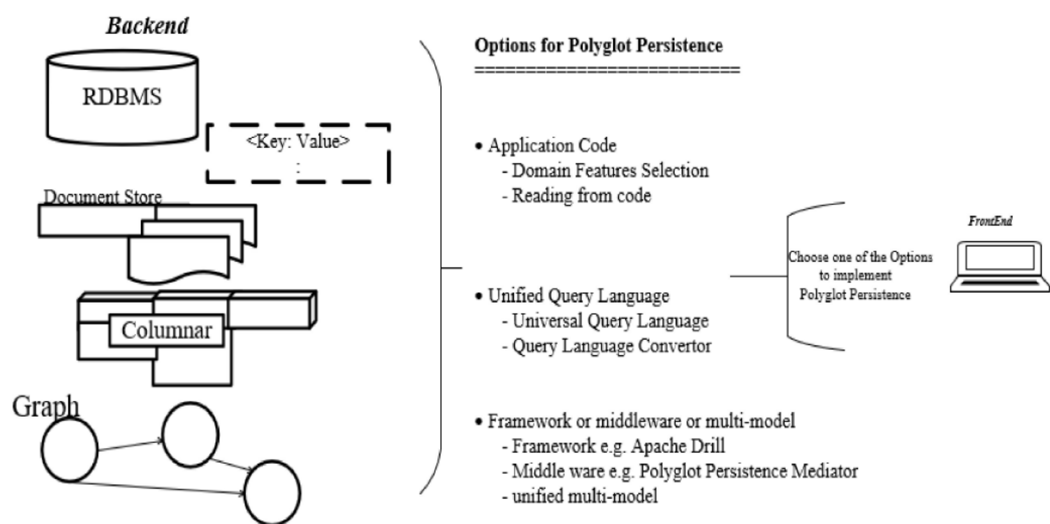


Figure. 2.2: Approaches to Unified Query System (Khine & Wang, 2019:p.18)

A polyglot persistent approach as illustrated in Figure. 2.2 have three possible avenues to explore when pursuing a unified query solution. Application code follows a specific pattern for data-intensive systems thereby producing an artifact using a Domain-Driven Design (DDD) approach sourcing data from different storage mechanisms. DDD involves the careful selection of shared datasets or domains for multiple targeted databases for the purposes of querying. Based on this premise, it fits the concept of a polyglot system. However, it does not fully realise

the notion of a unified query system as any enquiries to the storage mechanisms are predefined. Any additions or modifications to its query capabilities requires the application source code to be adjusted or refactored.

A Universal Query Language effectively condenses heterogeneous database schemas into a single component (Candel, Ruiz & García-Molina, 2022). The method relies on a well-defined design model upfront, detailing every aspect of targeted source schemas. The awareness of the source schema enables the universal query language to translate a query that is syntactically and semantically adjacent to the targeted storage system (Gobert, M., 2020; Kolonko & Müllenbach, 2020). Frameworks or middleware has been the most widely adopted selection for unified queries. It should be noted to the reader that there is a clear distinction between the two.

Frameworks for a unified query system concentrate strongly on the overall design aspect in order to deliver a well-suited solution. It comprises of a collection of libraries that are package in a unique way, providing a set of tools or templates to execute query commands (Ramadhan et al., 2020). The middleware consists of several independent components, each one with a clear set of objectives to satisfy. The infrastructure of middleware systems are arranged to facilitate data exchange with remote or local systems via its interfaces (Zhang et al., 2021). The operational boundaries are clearly defined for each component thereby reducing redundant functionality.

2.6 Design And Architecture

The open-ended nature of a polyglot persistent systems permits originators to developed a unified query platform using a range of methods and techniques (Glake et al., 2022). However, unified query solutions tends to follow similar design and architectural patterns. Despite the fact that there are a number of terminologies for these design and architectural segments, this study attempts to simplify it by categorising these interacting layers as follows:

- Data Layer
- Connector Layer
- Intermediatory Layer
- Application Layer

Each layer has number of components associated with it is responsible of a set number of operations. The data layer represents the link to the local or remote heterogenous repositories. This layer may in some instances detail the target repositories domain and schema information which aids in generating native queries. The repositories are loosely coupled within the data layer limiting the exposure of query solution to adverse effects of a storage model. Thus a faulty storage mechanism will not break the entire unified query system. This also provides the storage engines with similar models the ability to be easily swapped in and out should the need arise making it more scalable.

The connector layer serves as an adaptor which permits the communication between the targeted databases and client application (Glake et al., 2022). It's a generic class containing interfaces for the supported storage systems to facilitate the exchange of data. The connecting layer is the bridge between the intermediatory and data layers. The intermediatory is essentially the middleware that make a unified query possible. Within this layer the components are primarily segmented into the semantic and syntax rule modules forming the basis of the query translation module (Zhang et al., 2021). It accepts a query command as input, validates it via its rules and proceeds to translate it to a well-form native query. Subsequently sending the native query to the data layer via the connector. Finally, the application layer is a simple interface that communicates with the middleware (Koutroumanis et al, 2021). It permits the passage of query inputs to attain the desired the result set.

2.6.1. Frameworks and Models

An important step in developing a unified solution is related to the design choices made right at the beginning of the development life cycle. Managing unstructured data across heterogeneous becomes exponentially more complex due to a limited amount of applicable design choices. Hewasinghage et al., 2021 posits that this area is still in its infancy and requires further research. Nevertheless, it is imperative to select the ideal design implementation depending on the uses case.

2.6.2. Resource Description Framework

A widely used approach to unify the assortment of NoSQL data storage models lies within the Semantic Web technology (Endris, 2020; Santana & Mello, 2020). Specifically, the Resource Description Framework (RDF) as it is able to represent data on the internet effectively. Its central objective is to interconnect heterogenous data thus providing a key component within the unified query model. RDF is able to capture the format of various data models, the traditional relational and NoSQL models alike.

This is achieved by organising dispersed data as a whole via a concept called triples which codifies a statement or raw data as subject, predicate and object (Khan et al., 2019; Endris, 2020; Cox et al., 2020). Consider the following statement : "A student has a residency on campus", a student being the subject, has a residency being the predicate and finally campus being the object. The subject and object are considered as unique entities, while the predicate describes the strength of the relationship.

Triples are categorised into three query patterns; namely star, chain and complex (Cox et al., 2020). The star pattern is characterised by its single adjacent relationships with surrounding nodes. Chain patterns uses a series of joins between nodes i.e. a nodes manifests as "friend-of-a-friend". Finally a complex pattern is a combination of star and chain patterns. These concepts are deeply rooted in the mathematical discipline of graph theory whereby vertices and edges are used to describe the ontologies (Davoudian, Chen & Liu, 2018). The available patterns RDF offers, enables the structures of data to change over time without changing the

consuming application. This feature makes RDF an ideal design choice for unified query solutions.

2.6.3. Save Our Systems

The SOS model originally proposed by Paolo Atzeni, Francesca Bugiotti, and Luca Rossi (2012) to permit unified access to key-value, document and column stores data stores. It provides an interface to access Redis, MongoDB and HBase storage mechanisms facilitating simple insert, get and delete operations.

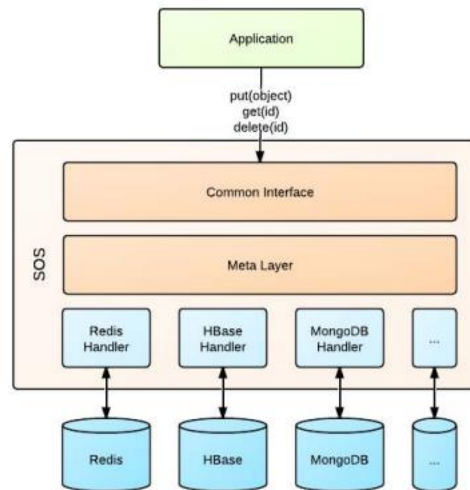


Figure. 2.3: SOS Architecture (Atzeni, Bugiotti & Rossi, 2012)

The solution is based on the meta-model construct, a data model for metadata utilised to describe external structures (Hewasinghage et al., 2021). It contains rules governing how to construct a data model or a domain for enquiry. The system is divided into three main modules, a common interface that serves as the starting point for engagement (Atzeni, Bugiotti & Rossi, 2012). The meta-layer module contains a repository of schema describing of the data models. Finally, it has separate handlers for responsible for interfacing with the supported NoSQL databases. The interactions between the modules represents an holistic view of the physical storage for the underlying NoSQL databases.

2.6.4. NoSQL Abstract Model

The NoAM metamodel is a design method for supporting NoSQL database that makes provisions for aggregated data models (Atzeni et al, 2020). NoAM is developed by the same creators who proposed the SOS model. Based on the shortcomings of SOS, NoAM was designated to improve on scalability, performance and consistency (Candel, Ruiz & García-Molina, 2022). The methodology establishes a conceptual depiction of a common data domain. The data models are built using a UML class diagram denoting the static nature of interested entities, its values and relationships. This information eventually forms part of the NoAM repository. NoAM catalogues shared attributes of the data across the heterogeneous NoSQL storage systems. Any variations are offset by the abstraction layer obscuring any finer details

in a particular domain. The abstract layer serves as the gateway to features of the underlying storage models enabling making a unified query possible.

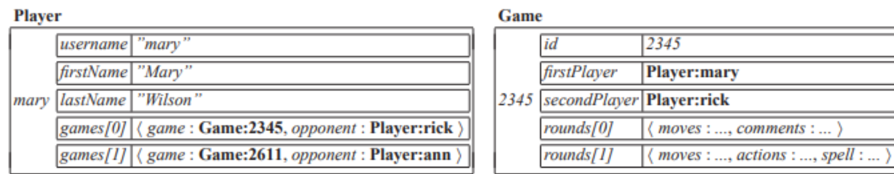


Figure. 2.4: Example of Structural Layout (Atzeni et al, 2020)

NoAM contains some key concepts which permits it to realise unifying heterogenous storage systems (Hewasinghage et al., 2021). The model contains a set of named collections within a set of blocks. The blocks within the collection is key, identified by a unique value. Nested within each block contains a set of entries with an identifier associated with a simple or complex value. Each of the forementioned concepts, as illustrated in Figure. 2.4, can be viewed as set of data models within another set of data models. Each of the data models are associated with identifiers thus enabling data access on various levels of granularity.

2.6.5. U-Schema Data Model

U-Schema's are representative of physical schemas for a given set of data model for disparate storage mechanisms (Candel, Ruiz & García-Molina, 2022). It aims to fully embody the structural dimensions across heterogenous NoSQL and relational SQL databases. The metamodel is heavily influenced by the patterns in data; therefore it classifies the organisation of data in two clusters referred to as its schema types.

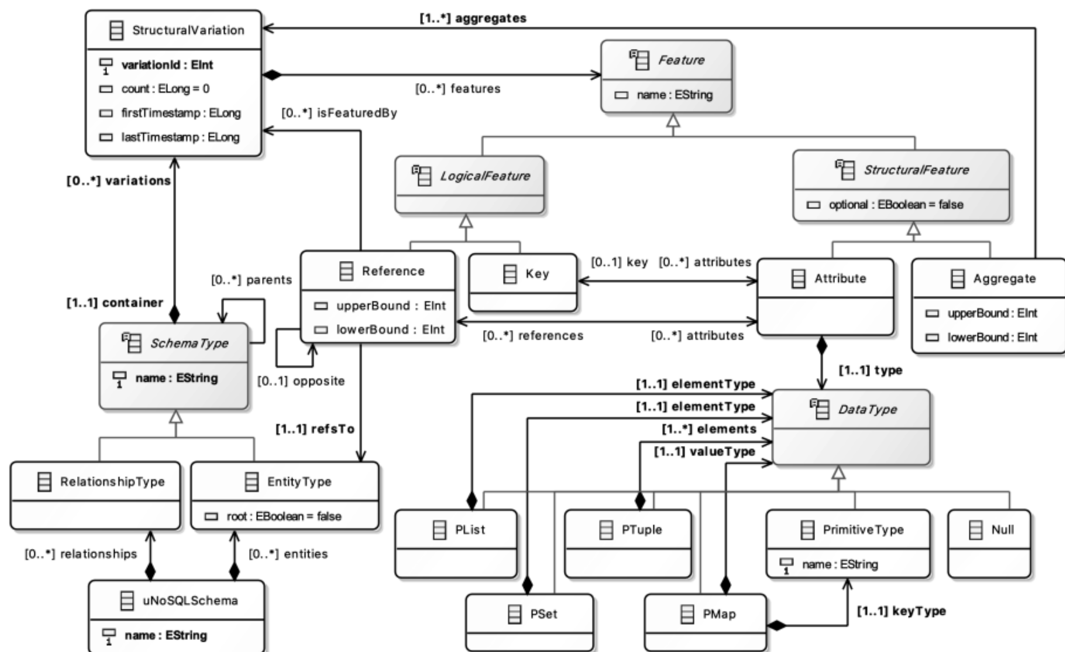


Figure. 2.5: U-Schema Metamodel (Candel, Ruiz & García-Molina, 2022)

Firstly, data where aggregation is prevalent utilises the aggregate-based model. The aggregate model are idyllic for key-value, document and column data store. As in the case of NoAM, every

aggregation is identified with unique keys (Atzeni et al, 2020). Secondly, in data that forms cyclic relations, the models systematically arranged as graph objects whereby a directed edge signifies the relationship. The U-Schema model principally is the amalgamation of entities type which represents data aggregation while relationships types are indicative of graph data. The structural prominence of documented schemas enables heterogenous data to be queried.

2.7 Polyglot Systems

Polyglot query systems has gained a substantial amount of traction in research years (Tan et al., 2017; Candel, Ruiz & García-Molina, 2022). This is particular accurate given the current climate, as data is ubiquitous. The number of unified query systems, while adhering to similar high-level architectures, each variation encompasses a unique class of problems which it aims to address (Kolev et al., 2016; Oussous et al., 2018; Roy-Hubara, Shoval & Sturm, 2022). The differences lies within the variety of approaches, methods, principles and technology instantiations to satisfy the intended use cases.

2.7.1. BigDawg

BigDAWG is an open source polyglot solution, developed by researchers at MIT with purpose of facilitating queryable interface for Apache Accumulo, PostgreSQL, Myria and SciDB databases (Glake et al., 2022). Researchers at MIT states that a polyglot system aims to "harness the relative strengths of underlying DBMSs" in order to effectively process data (Gadepally et al., 2016:p.2). The solution subscribes to three types of data models; key-value, relational and array stores. The architectural topology entails four separate layers:

- Application
- Middleware and API
- Islands
- Storage Engines

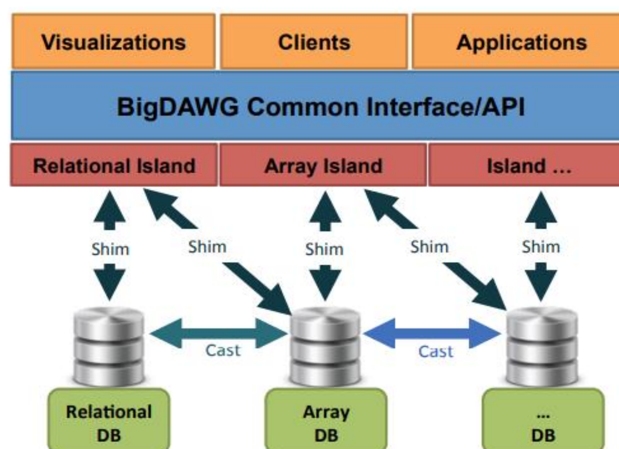


Figure. 2.6: BigDawg Architecture (Gadepally et al., 2016)

As illustrated in Figure. 2.6, the BigDawg architecture is more geared toward query processing instead of query construction. Its goal is to utilise key features in order to achieve the best possible performance and most complete result set. To realise this goal, the architecture has

features such as islands, shims and cast to support this idea (Gadepally et al., 2016; Glake et al., 2022). An island is associated with a data model and set query language features for the store engine it intends to support. A shim serves as the communication bridge between the island and the storage engines. A cast facilitates data migration from one storage engine to another.

The API routes enquiries to the middleware which is responsible for query execution as well as data migration via casts (Kolev et al., 2016). The middleware holds several modules such as the query planner, performance monitor and executor which validates the semantic correctness of queries and routes it to the relative storage mechanism for execution. It holds a history of past queries performances that routes the query workloads to the ideal storage engine (Tan et al., 2017). The BigDawg solution holds an important feature to ensure elasticity within the unified query solution where consumer can specify which islands to target. This decision is guided by the data that exists in the collection of storage engines.

2.7.2. Heterogeneous Middleware by Zhang et al

A solution brought forward by Zhang et al. (2021), proposed the use of middleware to execute queries on several heterogeneous databases via a single interface using standard SQL syntax. It's segmented architecture isolates the initial query from the actual targeted queries via an abstract syntax tree responsible for determining if the initial query conforms to the specificity of the respective heterogenous databases. The article mentions that the middleware supports a pluggable interface for any new data source but it does not elaborate how this will affect the abstract tree and computing layer.

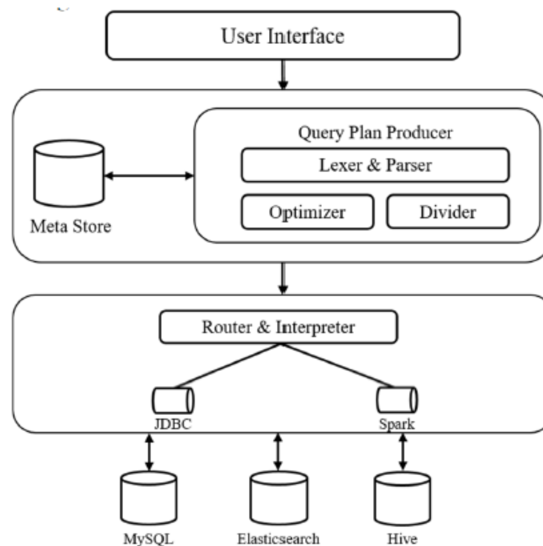


Figure. 2.7: Unified SQL Query Middleware Architecture (Zhang et al., 2021)

The middleware provides consists of a syntax parsing layer, computing engine and finally a data layer. The syntax layer validates a unified query based on the customer abstract syntax tree. Native queries are subsequently generating based its meta store which is delegated to the

computing engine for execution on the data layer. The proposed solution is limited to select queries and subsequently does not address evolving schemas. Additionally the middleware utilises wildcards which is not ideal as this would cause runtime issues stemming from datatype mismatches. It fails to account for schema evolution and does not address everchanging real-world problems as the solution invariability lacks scalability.

2.7.3. NoDA

NoDA is a lightweight implementation that serves as an abstract layer between the application and the targeted NoSQL databases which comprises of MongoDB, HBase, Redis, Neo4j (Koutroumanis et al., 2021). The middleware provides a generic set of operators such as sorting, filters and aggregations in its pursuit to efficiently execute queries using the Apache Spark open source data analytical framework. While NoDA is considered to be a polyglot implementation, it's less complex as it decouples (i.e. uses a third part tool) the rule engine from the abstract layer in order to validate syntax and semantics of the unified query.

Furthermore, even though the middleware supports the four primary categories of NoSQL data models, the system is only able to query one underlying database at any given time. This is made abundantly clear by the authors since the prototype is more concerned with the system's ability to access data through its connector. In addition it lacks a translation engine responsible for generating a multiple native queries in a single action including an intelligent query routing mechanism to defer the executable query to the respective heterogenous NoSQL databases.

2.7.4. Translator Query

The Translator Query Language (TranQL) is one such a solution presented by Cox et al. (2020) that federates Biomedical ontologies within a framework. The paper basis it findings on real world case studies using natural language to map to TranQL and subsequently targeted queries on various graph data models. The Translator KGS API, an integral part of the framework, which uses the shared schema RDF concept expressing a query as Biolink data model, i.e. hierarchical medical ontology on a high level, mapping a network of knowledge graphs as a coherent whole. This ultimately forms the basis for TranQL as a uniformed query pattern by interconnecting federated knowledge graph data models though curated links across entities.

2.7.5. Apache Drill

Apache Drill is fully distributed open-source software framework intended for large scale analysis on data-intensive applications (Hewasinghage et al., 2021). It focuses on processing a huge data sets in an efficient manner by executing tasks in parallel. The Apache Drill solution is influenced by in-memory data representation in JSON and Parquet format for fast data manipulation operations. Furthermore, its MPP query engine compiles and recompiles data queries on the fly to maximize performance relying on parallelism (Tan et al., 2017).

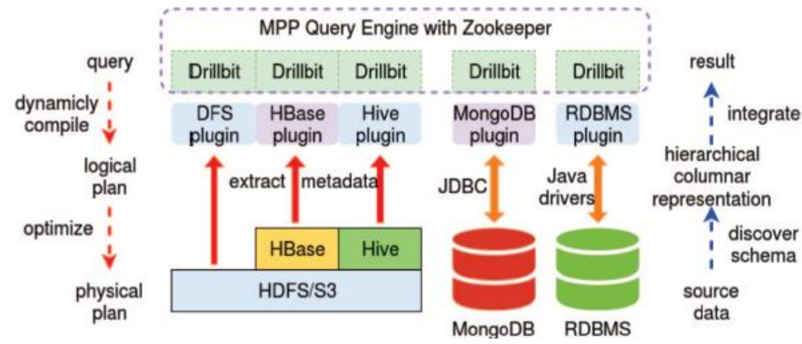


Figure. 2.8: Unified SQL Query Middleware Architecture (Tan et al., 2017)

The distributed nature of this polyglot solution results in a number of clusters containing a set of nodes that hold different data models (Khine & Wang, 2019). The Apache Drill's support data models are access through a similar mechanism to the BigDawg's implementation. Instead, of islands it uses plugins to connect to the different storage engine and file systems via the Drillbit component (Glake et al., 2022). Drillbit is a background component, orchestrating the optimal execution query plan for execution. The query executions are partially rendered on an execution tree and brought into memory.

2.7.6. CloudMdsQL

CloudMdsQL is considered more of a multistore system, capable of query multiple databases via its SQL-Like unified query construct (Kolev et al., 2016; Glake et al., 2022). As stated in Section 2.5, the terminologies used within the polyglot sphere are used interchangeably. CloudMdsQL supports relational, NoSQL and HDFS storage mechanisms. The originality introduced by the CloudMdsQL stems from its design traits which exploits each of its supported heterogeneous data store's built-in features (Koutroumanis et al, 2021). It's abstraction layer catalogues the supported data stores semantics rules enabling native queries to be optimized. This permits the construction native queries through a relation query framework for targeted executions. The embedded invocation results are converted into a intermediatory table for distributed processing.

2.8 Critical Evaluation of Unified Query Systems

It should be stated that this literature review does not cover all unified solutions, as attempting to describe all possible solutions is not the objective. Instead this study introduces the reader to the differentiating components that makes up these solutions. Research papers proposing unified query solutions understandably give focus on the overall utility of the artifact. A lot of emphasis to given to certain practicalities such as query workloads, indexing and partitioning i.e. query processing (Khan et al., 2019; Endris, 2020).

At this point it should be apparent that using a polyglot approach, there is an immediate trade-off in functionality of the native databases full features. Especially in the case of RDF solutions using federated knowledge graphs, where it's been well documented that a large amount of

ontologies irrespective of the triple configuration causes huge performance degradation (Cox et al., 2020). It requires data to be joined on an abstracted layer.

Native processing on the same dataset should always perform better when compared to non-native since there is no overhead to deal with. Hence the transversal capabilities is limited as the full feature of the knowledge graph cannot be exploited. This echoes the fact that scrutinization of querying processing in its entirety is given preference, instead of research on specific elements of the query process (Santana & Mello, 2020).

An important feature brought to the reader's attention relates to the visibility of the underlying heterogenous database structures. While number of approaches exists such as GaV and LaV to support data modelling of heterogenous structures, it is not made abundantly clear how the inevitable mismatches between the various storage mechanism should be handled (Candel, Ruiz & García-Molina, 2022). The mismatches comes in the form of query syntax and semantics, supported operations of the individual database and of course data discrepancies.

A good starting point is to conceptually split the structure and behaviour for a unified solution (Roy-Hubara, Shoval & Sturm, 2022). The behaviour model may catalogues the features of the native database thus providing a unified query system information on which operations are permitted. The structural model provides the designer an opportunity to consolidate naming, attribute, precision and domain conflicts. This ensures the data queried in the heterogenous databases are understood in the same way.

The unified solutions described in this study are aimed to satisfy different use cases. Apache Drill for example is proficient in processing large amounts of data for analysis. Its requires a powerful machine as it load data in memory for fast retrieval (Glake et al., 2022). CloudMdsQL and BigDawg on the other hand uses the full extent of the supported databases native features to process data. Thus providing users with more capabilities. TranQL is federated query system for Boilink data using a topology of graph stores. Each of these solutions embeds an collection of individual isolated components that targets the supporting databases . The component are operates in silos serving intermediatory between the middleware layer and the database, with the exception of BiGDawg where data integration between silos are possible.

The other solutions are less intricate following the basic principles described in section 2.6. These prototypes focuses more on the query construct (Koutroumanis et al., 2021; Zhang et al., 2021), which this study aim to achieve. However, in both instances, the authors do not quite elaborate extensively on the knowns and unknowns of unified queries. In general, there are not a lot of focus is given to the construction of an abstract queries and how it may be decomposed to fit the needs of a key-value, column, document or graph data models via a single act. Moreover, when considering RDF which is mainly concerned of classifying shared entities across heterogeneous storage data models, it loses the granular data specific to a particular storage which may in fact prove useful.

2.9 Systematic Literature Review

This study embarked on a systematic literature review (SLR) commonly classified as a secondary study assisting researchers with finding, scrutinizing and cataloguing the applicable academic papers within a specific domain (Kitchenham et al., 2010). The aim of conducting the SLR for this research study is to employ a scientific approach without bias. The SLR is repeatable process engaged in advancing knowledge based on previous research endeavours (Okoli & Schabram, 2010).

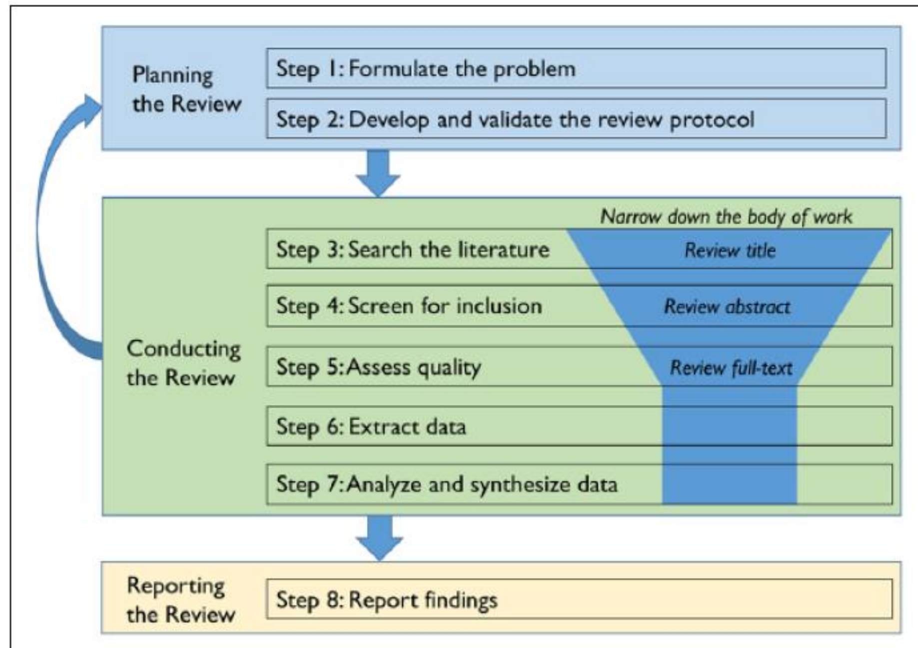


Figure. 2.9: Systematic Literature Review process (Xiao & Watson, 2019)

In order to systematically extrapolate knowledge from research papers on the design and development of unified query systems, this research project adheres to the eights steps proposed by Xiao & Watson (2019) to conduct the SLR. The steps are further organised into three category planning the review, conducting the review and reporting the review.

2.9.1. Planning the review

Authors agrees that a study's research questions are the driving force for initiating the systematic literature review process (Kitchenham et al., 2009; Xiao & Watson, 2019). The goal is to seek answers to the question posed in this study. Xiao & Watson, 2019, further cites a paper by Cronin, Ryan & Coughlan (2008) that cautions against formulating systematic literature questions that a too open-ended as this will result in large amount of data returned. Therefore the SLR questions constructed in this study strived to be specific and relevant to the problem domain as shown in table 2.2, to reduce the amount of irrelevant and duplicate data.

Table 2.2: SLR Questions

No.	Questions
SLRQ1	What are the existing guidelines for developing unified query platform for a polyglot system?

SLRQ2	What are the typical design and architectural principles for unified query platform for a polyglot system?
SLRQ3	What the different approaches used to develop a unified query platform?
SLRQ4	How can an optimal unified query system be achieved?

SLR Protocols

To establish the required research rigor, the study engaged in defining the necessary protocols to remove research bias while increasing the reliability of the review process (Kitchenham et al., 2009). Conclusively documenting the search strategy, the inclusive and exclusive criteria, quality assessment and screening procedures provides a solid foundation for extracting the correct data related to unified query platform systems.

Search Strategy

The search strategy utilised reputable online academic search engines, scholarly databases and digital libraries to source the relevant articles, journals and conference papers (Table 2.3). The search process predominantly used google scholar and the university's (CPUT) library to access articles that's not available without a subscription.

Table 2.3: SLR Search strategy

Academic Repository	Location
Google Scholar	https://scholar.google.com
IEEE Explore	https://ieeexplore.ieee.org
ACM Digital Library	https://dl.acm.org
Springer	https://www.springer.com
Science Direct	https://www.sciencedirect.com

The study conducted the search procedure in a manual fashion. While this process was tedious it provided a measure of control whereby enabling the researcher to dynamically refine the search criteria. Xiao & Watson, 2019, cites a study conducted by Oppenheim & Rowland (2008), found that google scholar when compared to other academic databases out performed its counterpart by providing reliable and relevant data. This paper included academic literature that was cited by other papers and reviewed authors contributions to determine the validity of the academic works.

Search Terms

As denoted in section 2.6.1, there are certain aspects of the unified query platform domain that is still in its infancy stage. The search terms defined for this study required varying and interchangeable terminology. The preliminary research process indicated that certain keywords are often loosely used hence adjustments has to be made to ensure that the appropriate papers were returned.

Table 2.4: SLR Search terms

#	Keywords
K1	Polyglot

K2	Polystore
K3	Middleware
K4	Unified Query
K5	Global Query
K6	Non-relational Schemas
K7	Unified Schemas
K8	Non-Relational
K9	Schemas
K10	Big Data
K11	NoSQL
K12	Query(ies)
K13	Data Model
K14	MetaModelling
K15	Review
K16	Survey
K17	Stores
K18	Abstract Syntax Tree
K19	Text-Based
K20	Heterogeneous
K21	System

The researcher applied the following search combinations to retrieve the relevant literature on unified query platforms:

- S1 – [K1,K2,K3] and [K4,12] and [K5]
- S2 – [K1, K6] and K3 and [K11]
- S3 – [K7] and [K1, K6] and [K10]
- S4 – [K8, K9], [K4, K9] or ([K13] or [K14])
- S5 – ([K15 or K16], K10), ([K11, K17]) and [K1, K6]
- S6 – [K19, 18]
- S7 – [K4, K12] and [K20, K11]
- S8 – [K2, K10, K21]

In pursuit of ensuring the relevant literature was retrieved from academic sources given the state of unified query domains, the key combinations was appended to create the ideal search combinations. This was done specifically since unified query systems has not matured when compared other systems in the engineering domain such as SQL DBMS.

2.9.2. Conducting the Literature review

The initial search was conducted in a crude manner which returned many results irrelevant to the research question established (Xiao & Watson, 2019). Hence the search terms were refined, executing the search combinations. To ascertain a reliable and complete collection of academic literature was given the necessary attention, the researcher additionally conducted a backward

search as well. This was achieved by identifying references cited by the research papers retrieved from the search process discuss in section 2.9.1.

Literature Search Results

Table 2.5 list the search results obtained from the search strategy which amount to 43 academics papers.

Table 2.5: SLR Results

Year	ACM Digital Library	Google Scholar	IEEE Explore	Springer	Science Direct
2012	1				
2013			2		
2014			1		
2015		2			
2016			2	1	
2017		2	1		
2018	1	1	2	4	1
2019	1	2	3		
2020	1		4	1	1
2021	1	3			
2022		2		1	2

Academic Inclusion

The process of selecting the relevant research papers on developing a unified query included academic literature on the foundations of big data, the various NoSQL databased that were developed to encapsulated these large amount of data and finally the design and principles on unified query systems. The study efficiently determined the articles to include by initially scrutinizing the abstracts and subsequently refined the inclusive list by assessing the quality of the articles (refer to Assessment Quality section).

Table 2.6: Criteria for inclusion

#	Criteria
C1	Primary studies where data was collected first hand.
C2	Secondary studies that review the current landscape of big data technologies.
C3	Approaches to design and architectural practices for polyglot approaches.
C4	Proposed and Propriety unified query solutions.
C5	Data modelling techniques for a unified query platform.
C6	Text-based query methods for a natural language.

Table 2.6: Criteria for exclusion

#	Criteria
E1	Research papers that was published before 2012.
E2	Research papers that was published with no citations.
E3	Research papers that does not address the SLRQ’s

E4	Research papers written in other languages.
E5	Research papers that requires a paid subscription

Quality Assessment

The assessment of quality was performed by reviewing the full text of the research papers identified in the academic inclusion step. The valuation step adopted the standard approach used to exercise an "internal validity" check based on a score for the inclusive papers (Xiao & Watson, 2019:p106).

Table 2.7: Quality Assessment Question and Scoring

#	Enquiry	Yes	Partial	No
QA1	Is the aim of research study clear?	1	0.5	0
QA2	Is the research methodology defined?	1	0.5	0
QA3	Is the context of the study relevant in relation to the problem domain?	1	0.5	0
QA4	Is the study research outputs valid and reliable?	1	0.5	0
QA5	Is the paper's findings generalized to the broader population?	1	0.5	0

This academic enquiry process was decoupled from this study's DSR methodological approach. The ideal answer to the all quality assessment questions, except QA4, for this study is score is 'Yes'. This classification applied to 'Partial' ratings as the minimum criteria for the study to be included in the reporting process. The desired answer to QA4 was an emphatic 'No'. This study ignored all research papers where the validity of the study was in question. Even in cases where it met the assessment criteria of the other categories.

Data extraction

The meta-analyses conducted on the academic papers were concluded by iteratively refining the search criteria and evaluating it against the system literature review research questions. Table 2.8 catalogues the list of academic papers that meets the criteria, performed a full text analysis on the refined data results. The table also includes data as a result of performing backward searches by research paper in the initial search activity.

Table 2.8: SLR Results

#	Authors	Year	Title	
P1	Glake, D., Kiehn, F., Schmidt, M., Panse, F. & Ritter, N.	2022	Towards Polyglot Data Stores--Overview and Open Research Questions	S1
P2	Tan, R., Chirkova, R., Gadepally, V. & Mattson, T.G.	2017	Enabling query processing across heterogeneous data models: A survey	
P3	Hewasinghage, M., Abelló, A., Varga, J. & Zimányi, E.	2021	Managing polyglot systems metadata with hypergraphs	
P4	Lindström, O.P.	2021	Integration of SQL and NoSQL database systems	

P5	Khan, Y., Zimmermann, A., Jha, A., Gadepally, V., D'Aquin, M. & Sahay, R.	2019	One size does not fit all: Querying web polystores	
P6	Santana, L.H.Z. & Mello, R.D.S.	2020	Persistence of RDF Data into NoSQL: A Survey and a Unified Reference Architecture.	S2
P7	Kolonko, M. & Müllenbach, S.	2020	Polyglot persistence in conceptual modeling for information analysis.	
P8	Košmerl, I., Rabuzin, K. and Šestak, M	2020	Multi-model databases-Introducing polyglot persistence in the big data world.	S3
P9	Candel, C.J.F., Ruiz, D.S. & García-Molina, J.J.	2022	A unified metamodel for nosql and relational databases.	S4
P10	Banerjee, S., Bhaskar, S., Sarkar, A. & Debnath, N.C.,	2021	A unified conceptual model for data warehouses.	
P11	Vajk, T., Fehér, P., Fekete, K. and Charaf, H.	2013	Denormalizing data into schema-free databases.	
P12	Davoudian, A., Chen, L. & Liu, M.	2018	A survey on NoSQL stores.	S5
P13	Khine, P.P. & Wang, Z.	2019	A review of polyglot persistence in the Big Data world.	
P14	Oussous, A., Benjelloun, F.Z., Lahcen, A.A. & Belfkih, S.	2018	Big Data technologies: A survey.	
P15	Roy-Hubara, N., Shoval, P. & Sturm, A.	2022	Selecting databases for Polyglot Persistence applications.	
P16	Atzeni, P., Bugiotti, F., Cabibbo, L. & Torlone, R.	2020	Data modeling in the NoSQL world.	
P17	Kazanavičius, J., Mažeika, D. and Kalibatienė, D.	2022	An Approach to Migrate a Monolith Database into Multi-Model Polyglot Persistence Based on Microservice Architecture: A Case Study for Mainframe Database.	
P18	Zhang, M.	2020	A survey of syntactic-semantic parsing based on constituent and dependency structures.	S6
P19	Duracik, M., Hrkut, P., Krsak, E. & Toth, S.	2020	Abstract syntax tree based source code antiplagiarism system for large projects set.	
P20	Koutroumanis, N., Kousathanas, N., Doulkeridis, C. & Vlachou, A.	2021	A demonstration of NoDA: unified access to NoSQL stores.	S7
P21	Zhang, H., Zhang, C., Hu, R., Liu, X. & Dai, D.	2021	Unified SQL Query Middleware for Heterogeneous Databases.	
P22	Ramadhan, H., Indikawati, F.I., Kwon, J. & Koo, B.	2020	MusQ: A Multi-store query system for iot data using a datalog-like language.	
P23	Amghar, S., Cherdal, S. and Mouline, S.	2019	Data integration and nosql systems: A state of the art.	
P24	Fathy, N., Gad, W. and Badr, N.	2019	A unified access to heterogeneous big data through ontology-based semantic integration.	

P25	Gadepally, V., Chen, P., Duggan, J., Elmore, A., Haynes, B., Kepner, J., Madden, S., Mattson, T. & Stonebraker, M.	2016	The BigDAWG polystore system and architecture.	S8
P26	Maccioni, A. and Torlone, R.	2018	Augmented access for querying and exploring a polystore.	
Academic literature retrieved via a backward search				
P27	Gobert, M.	2020	Schema Evolution in Hybrid Databases Systems.	
P28	Kolev, B., Bondiombouy, C., Levchenko, O., Valduriez, P., Jimenez-Péris, R., Pau, R. & Pereira, J.	2016	Design and implementation of the CloudMdsQL multistore system.	
P29	Cox, S., Ahalt, S.C., Balhoff, J., Bizon, C., Fecho, K., Kebede, Y., Morton, K., Tropsha, A., Wang, P. & Xu, H.	2020	Visualization Environment for Federated Knowledge Graphs: Development of an Interactive Biomedical Query Language and Web Application.	
P30	Endris, K.M.	2020	Federated Query Processing over Heterogeneous Data Sources in a Semantic Data Lake.	
P31	Guo, J., Liu, Q., Lou, J.G., Li, Z., Liu, X., Xie, T. and Liu, T.	2020	Benchmarking meaning representations in neural semantic parsing.	
P32	Yang, X., Zhang, X. & Tong, Y.	2022	Simplified abstract syntax tree based semantic features learning for software change prediction.	
P33	Öztürel, İ.A.	2022	Cross-Level Typing The Logical Form For open-domain semantic parsing.	

Data synthesis

Table 2.7. shows the scores how the researcher scored each paper in relation to the quality assessment criteria.

Table 2.9: Search terms

Paper	QA1	QA2	QA3	QA4	QA5	Total Score
P1	1	0.5	1	0.5	0.5	3.5
P2	0.5	0.5	0.5	1	1	3.5
P3	1	1	1	1	1	5
P4	0.5	1	0	0	0.5	2
P5	1	1	0.5	0.5	1	4
P6	1	0.5	0.5	1	1	4
P7	1	0.5	1	1	1	4.5
P8	1	1	1	1	1	5
P9	1	1	1	1	1	5
P10	1	0	0	0	0	1

P11	1	1	1	1	1	5
P12	1	1	1	1	1	5
P13	1	1	1	1	1	5
P14	1	1	1	1	1	5
P15	0.5	0.5	0.5	0	0	1.5
P16	1	1	1	1	1	5
P17	0.5	0	0	0	0	0.5
P18	1	1	0.5	1	1	4.5
P19	1	0.5	0.5	0.5	0.5	3
P20	1	1	1	1	1	5
P21	1	1	1	1	1	5
P22	1	1	1	1	1	5
P23	0.5	0	0.5	0	0	1
P24	0.5	0.5	0	0.5	0.5	2
P25	1	1	1	1	1	5
P26	1	0.5	0	0	0.5	2
P27	1	1	1	0	0	3
P28	1	1	0.5	1	1	4.5
P29	1	1	0.5	1	1	4.5
P30	1	1	0.5	1	1	4.5
P31	1	1	1	1	1	5
P32	1	1	0.5	1	1	4.5
P33	0.5	0.5	0.5	0.5	0.5	2.5

2.9.3. Reporting the review

The broad search results consisted of a number of research endeavours amounting to 54 academic papers where matches were found in the title. However upon further review the list was condense to 43 articles. The researcher proceeded scrutinise the abstracts of the results, based on the system literature review questions, which then further narrowed do the amount of data to 33 research papers. A full text analysis was performed against the quality assessment rating defined in table 2.7. The research papers were scored 2.5 or higher was accepted, hence the final amount of paper resulted in 27 academic research papers on unified query platforms.

2.10 Summary

The aim of this chapter was to gain a better understanding of unified query solutions. The chapter draws the reader's attention back to the research questions and objectives. It progresses to address these elements of this study be assessing the related works on unified query systems. Particularly, the general make up and nuisances of existing theories and instantiations.

In part, a systematic literature review was conducted to establish the research rigour required for the literature review independent of the research methodology for the study (Okoli & Schabram, 2010, Xiao & Watson, 2019). This consisted of an eight step process, from

formulating the problem to applying a search strategy to reporting the final result set. Academic literature on unified query systems supporting different NoSQL data models it still very much in its infancy (Khine & Wang, 2019; Hewasinghage et al., 2021). A lot of factors are to be consider when partaking in such an endeavour. The researcher noted due to the level of maturity in the problem domain, key terms were often misused hence the preliminary results return an expected amount of data. Through careful planning and understand of the domain, through a number of iterations the data was restricted to a manageable amount.

Thus the systematic activity provided clarity and context for the study's problem. Through the literature analysis, the researcher discovered the certain papers are vague when expressing where advances are made and in which area of the query processing model it should apply. According to Santana & Mello (2020), many surveys relating to unified queries are constrained by artificial and antiquated benchmarks. More importantly, it is not clear how to effectively develop a unified query for the four NoSQL storage categories.

CHAPTER THREE : RESEARCH METHODOLOGY

3.1 Introduction

This chapter identifies, describes and justifies the research methodology employed in this study to achieve the research questions outlined in section 1.5. The research methodology is a set of scientific techniques and methods applied to a field of study (Khaldi, 2017; Mardiana, 2020). This study adopts a quantitative approach; since the impetus for this research is to generalize its findings to a broad spectrum within the unified query domain. Therefore the study's philosophical stance is that of positivism since the empirical evidence was gathered through an experimental prototype to reveal a degree of certainty by means of statistical information (Hevner et al., 2004). This approach is supported by Saunders et al., (2012:113) when articulating research from a positivists point of view.

"*If your research reflects the philosophy of positivism then you will probably adopt the philosophical stance of the natural scientist. You will prefer 'working with an observable social reality and that the end product of such research can be law-like generalisations similar to those produced by the physical and natural scientists' (Remenyi et al. 1998:32)*".

This research endeavour applies Design Science Research as its principle research strategy as it focused on the development of an experimental artifact. Hevner et al. (2004) states that a primary motivation for using DRS is to gain understanding and new knowledge of a problem domain through a novel artifact that is able to clearly demonstrate its application. Baskerville et al. (2018) posits that within a technological context; DRS may be used to build on existing knowledge of a particular area, thus enhancing the efficacy between humans and technology. Therefore DRS was selected as the research design choice for this study, as it is most appropriate since the objectives of the study was to design, construct and evaluate a working prototype.

3.2 Research Paradigm

A research paradigm is the philosophical ideology of how reality is observed (Khaldi, 2017; Alharahsheh & Pius, 2020; Mardiana, 2020). Saunders et al., (2012) acknowledges that this term is often used within a multitude of circumstances, potentially leading to misinterpretation of its purpose to novice researchers. The research paradigm embodies a set of theories and practices for engaging in a research endeavour which serves a guide to solving a particular problem domain (vom Brocke, Hevner & Maedche, 2020). Given a researcher's philosophical choice, it steers the research effort in selecting the appropriate practices ensuring the validity of the study when addressing the research questions. The research paradigm is predominantly predisposed into three viewpoints:

- Ontology
- Epistemology
- Axiology

These influencing factors innately dictates the practical considerations in a research field of study. More specifically it guides the research process as it correlates to how knowledge is attained (Saunders et al., 2012). As a result, the forementioned descriptions underpins the research strategy utilised in a study.

3.2.1. Ontology

The ontological viewpoint ponders on what constitutes as a factual phenomenon. It is concerned with how reality or the world is perceived. This branch of philosophy has two opposing aspects, objectivism and subjectivism (Saunders et al., 2012; Merwe, Gerber & Smuts, 2019). Objectivism is an unbiased reflection on a particular observation. The act may be independently substantiated be separate social actors in order for this reflexion to be truly deemed as objective. Whereas subjectivism is a belief system where social facts, based on a social perception, is deemed as factual.

3.2.2. Epistemology

Epistemology is a branch of philosophy interested in "what constitutes acceptable knowledge in a field of study" (Saunders et al., 2012:p.112). It's an entirely separate field of research contemplating the theoretical aspects of knowledge. Depending on a researcher's philosophical point of view, objective or subjective, the truth of reality is interpreted differently due to what it deems tangible and intangible (Alharahsheh & Pius, 2020). Objectivists are more concern with physical evidence to support their theories while subjectivist influences are based off the social perceptions of reality.

3.2.3. Axiology

Axiology is concerned with the value or validity of a research study. As Saunders et al., (2012) posited, it encompasses considerations made during the research process which includes aesthetics and ethics. Its interested in the way a social enquiry was conducted and characterising whether the value of the knowledge attained is good or bad.

3.2.4. This Study's Philosophical Position

The research's study is motivated by deductive reasoning therefore it relies on objectivism to ascertain unbiased facts. The physical evidence attained during the research process serves as the pillar to justify its findings and contribute to new knowledge.

3.2.5. DSR within the Research Paradigm

The role of Design Science Research within the research paradigm has evolved as more researchers have adopted this strategy in an effort to direct the research process (Peffers et al., 2020). A publication by Merwe, Gerber & Smuts (2019) discovered three philosophical viewpoints numerous authors holds when applying DSR to a study (Table. 3.1). Firstly, there has been consensus amongst researchers for DSR to be reimagined as an independent research paradigm (vom Brocke, Hevner & Maedche, 2020). These authors argues that DSR contributes affects the social settings through new pioneering artifacts. Hevner et al. (2004) on

the other hand proposes a pragmatic view to be adopted for DSR. The justification provided for this philosophical posture argues that DSR requires a synergy between theoretical and practical contributions in order to be considered valid a practice. Merwe, Gerber & Smuts (2019) further reinforces this idea by citing authors in support of this supposition by emphasising that empirical clarity can only be realised once in practice. The final school of thought which aligns with this studies viewpoint, remains grounded in the traditional paradigms such as positivism whereby DSR remains a strategy which is fortified by philosophical choices.

Table 3.1: Research Perspectives for DSR (Adapted from Vaishnavi, Kuechler & Petter, 2019)

	Ontology	Epistemology	Methodology	Axiology
Positivist	A single reality; knowable, probabilistic	Objective	Observation; quantitative, statistical	Truth; universal and beautiful; prediction
Interpretive	Multiple realities, socially constructed	Subjective	Participation; qualitative. Hermeneutical, dialectical.	Understanding: situated and description
DSR	Multiple, contextually situated alternative world-states. Socio-technologically enabled	Knowing through making.	Developmental. Measure artefactual impacts on the composite system.	Control; creation; progress (i.e. improvement); understanding

Hevner et al. (2004:p.4) reasons that DSR is a needed within the research paradigm as it embodies an innovative spirit to create ‘purposeful artifacts’. The origins of DSR stems from the natural sciences. Based on the positivistic paradigm made in this study, DRS is an appropriate strategic choice as it fits the natural scientists viewpoint. Conventionally, positivism is used to test a hypothesis (Khaldi, 2017). However, this study does not follow to this standard. Instead the philosophical assumptions are in part influenced by the data this study measures. In addition, as stated to the reader in Section 1.4, the overarching goal for this study was to create a decisive artifact that represents ideal practices when developing unified query systems. While this study does not seek to change organisational and human behaviour, it seeks to introduce a change in the environment which the forementioned stakeholders operates in (Pries-Heje, Baskerville & Venable, 2008; Baskerville et al., 2018), indirectly affecting these stakeholders. DSR thus solves important problems through a combination of design and natural science paradigms .

An important observation noted, DSR does not form part of the research onion framework proposed by Saunders et al., (2012). It is a widely used framework for ensuring a researcher

applies an effective research methodology based on their philosophical viewpoint. Mardiana (2020) argues DRS is excluded from the research onion as it was tailored more towards the needs of business organisations primarily concerned with human behaviour.

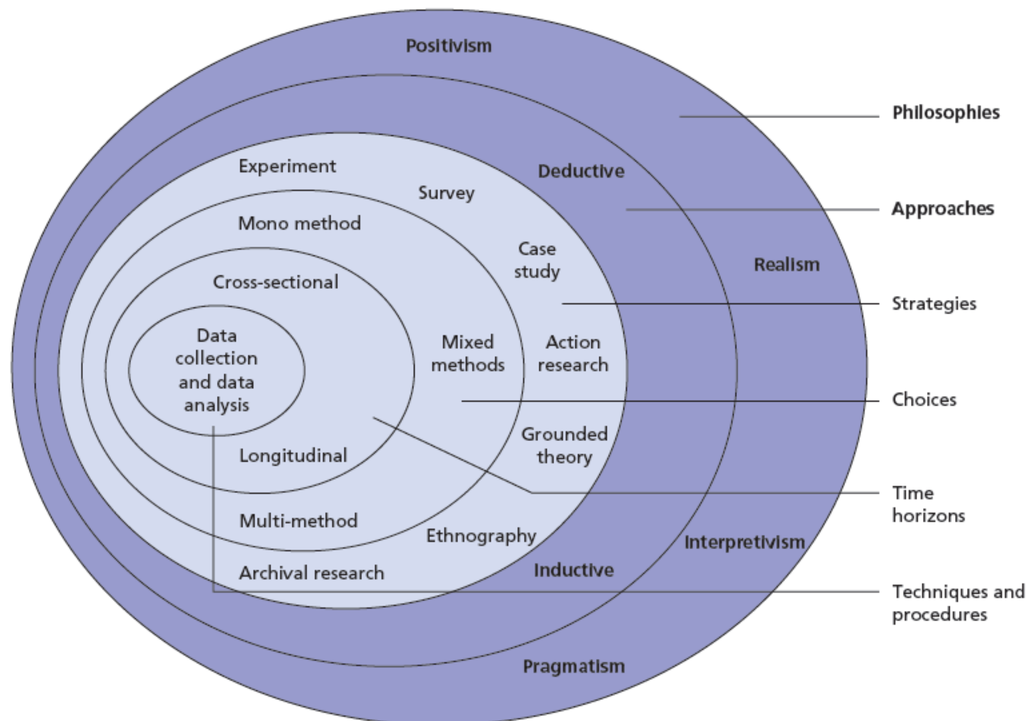


Figure. 3.1: Research onion

Considering that DSR research has made several contributions to the existing body of knowledge. This study can only speculate the reason for its exclusion from the research onion, perhaps due to its past failures or that the fact that DSR is fairly new when compared to traditional practices (Hevner et al., 2004). As stated in the beginning of this section, there has been many proposed frameworks and seminal papers advocating DSR and its position within the research paradigm. Merwe, Gerber & Smuts (2019) discussed how authors raised opposing views on the subject, each with its own theories of what constitutes a good DSR research endeavour. Perhaps due to this uncertainty it was excluded. Nevertheless, Mardiana (2020) contends that DSR should be included within the strategic layer of the onion (Figure. 3.1) in IS research. The research onion in part, together the DSR guidelines in section 3.3.2, was used to assist in the decision-making process, selecting the most appropriated techniques and methods to steer this research effort.

3.3 DSR as Strategy

The Design Science Research (DSR) is a problem-solving architype that creates knowledge on the design process and product concurrently (vom Brocke, Hevner & Maedche, 2020). It seeks to improve the artificial environment of a particular domain through innovative artifacts. A revealing trait innately associated with DRS, is that it is required to contribute to the existing body of knowledge through scientific means via the purposeful artifact (Hevner et al., 2004;

Baskerville et al., 2018; Merwe, Gerber & Smuts, 2019). The absence of this characteristic, will ultimately render any DSR research effort invalid. The knowledge contribution may affect existing theories and designs in two ways. Either it enhances the existing knowledge base or it may render it obsolete, thus introducing a tangible change in the environment under which it operates.

DSR is differentiated from other strategies in a unique way whereby the actual design is expressed as both a process and a product (Hevner et al., 2004; Pries-Heje, Baskerville & Venable, 2008; Baskerville et al., 2018). The process is articulated as an act while the product is stated as a sense. The culmination of the artifact can be viewed as a step by step guideline to construct impactful design theories and practices. However the value of the artifact needs to be assessed by internal controls governed by the research process model to determine if it is indeed impactful. This is determined by evaluating how deeply-rooted kernel theories and accepted principles are entrenched in the artifact (Khaldi, 2017). The degree of this alignment is indicative of its utility. In addition, the applied contributions provides further empirical evidence of its usefulness.

According to Rittel and Webber (1984) as cited by Hevner et al. (2004:p.6) considered design science as a “wicked problem”. At the time, this characterisation aptly defined design science research, as its methodology could not be clearly articulated. Rittel and Webber argued that design science projects relied too much on human reasoning for complex problems in an ill-defined environmental context. One might argue that this still exists to a certain extent given the diverse philosophical views on DSR by authors as cited by Merwe, Gerber & Smuts (2019). The flexible nature of developing an artifact using DRS integrally opens the research endeavour to criticism since research boundaries may be transgressed. As opposed to traditional research methods and techniques which are well-defined and may be easily assessed to determine the axiological output. To address this uncertainty many authors have proposed a research design science research process model (DSRP) to guide the research process (Peffers et al., 2020; vom Brocke, Hevner & Maedche, 2020).

3.3.1. DSR Process Model

As discussed towards the end of the previous section, the reader was introduced to the uncertainty that exists in DSR when assessing its utility in relation to research methods employed. Hence, the objective of a design science research model is to provide a mental model of research outputs at every stage in the research cycle (Peffers et al., 2020). This in turn forms the basis of the research methodology which operationalises DSR. It provides a full trace of the research activities performed offering guidance to researchers. This removes any facile elements of DSR, reinforcing the required research rigor.

The research process model applied to this study is founded on this fundamental building blockings of DSR, namely of awareness of problem, suggestion, development, evaluation and conclusion (Vaishnavi, Kuechler & Petter, 2019). It adheres to a set of prescribe guidelines as

described by Hevner et al. (2004) in relation to the proposed experimental prototype to strengthen the necessary research thoroughness. The build and evaluate loop serves as a template for conducting the development of the artifact to ascertain applicable new knowledge.

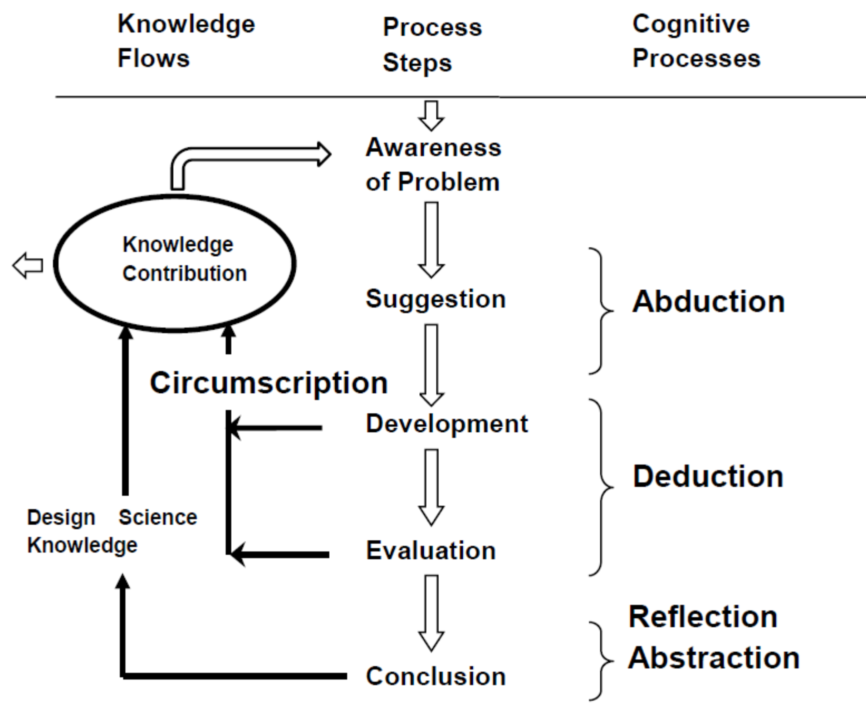


Figure. 3.2: Cognition in RSDP Model (Vaishnavi, Kuechler & Petter, 2019:p.59)

Knowledge contributions in any DSR cycle must always be at the forefront of the researchers mind as illustrated in Figure 3.2. The research endeavour always starts at the awareness of a problem based on gaps in the artificial environment (Peffers et al., 2020). This activity in the research cycle is always revisited due to the very nature of the problem. The initial suggestions posed to the problem will inevitably be inadequate confirming the fundamental trait of the research problem. Suggestions are drawn from literature whereby existing kernel theories are tied together to form a probable creative solution based on the researcher's motivation (Baskerville et al., 2018).

The first iteration of the development activity typically reflects these knowledge gaps as the research discovers which parts of the known knowledge base works after the evaluation activity. Therefore, an important part of the DSRP model, is the circumscription process. A logical technique employed to determined which parts of the knowledge base satisfies the gaps described in the problem (Vaishnavi, Kuechler & Petter, 2019; Peffers et al., 2020). Finally, the conclusion activity signals the end of the research cycle, reflecting on the observations made and generalising the applicable findings. Table 3.2 provides a summarised view of the research cycle.

Table 1.2: Adapted from DSRP Model Activities (Vaishnavi, Kuechler & Petter, 2019)

#	Activity	Description	Output
1	Awareness of Problem	The identification of the research problem which is goal-orientate i.e. problem-solving paradigm.	Formal or Informal Proposal
2	Suggestion	The envisioned solution inspired by the research problem.	Tentative design
3	Development	The tentative design is physically constructed. Note the novelty lies within the design not really in the construction.	Novel Artifact
4	Evaluation	The novel artifact assesses its utility via the identified variables in relation to the problem.	Performance metrics, Suggest a new design approaches or theories
5	Conclusion	Communicates empirical facts: new learnings, deviations from hypothetical predictions, limitations, justification for results	Report

In order to adhere to the research rigor described above for this study using DSR, existing literature on unified query solutions was scrutinize; making the necessary inferences to ensure a purposeful contribution to the existing body of knowledge via the experimental prototype (Hevner et al., 2004; Vaishnavi, Kuechler & Petter, 2019). A common theme that is expressed in numerous articles and journals on unified solutions, is simply the over reliance on human intervention due to its complexity. Thereby establishing the impetus for a DSR pursuit, in what Hevner et al. (2004:p.85) describes as the "human-machine problem-solving" system. Unified query solutions fits this description aptly as number of authors have demonstrated this phenomenon (Cox et al., 2020; Koutroumanis et al., 2021; Zhang et al., 2021).

3.3.2. DSR Guidelines

The research origins of design science is not rooted in the conventional IS research paradigms (Hevner et al., 2004; vom Brocke, Hevner & Maedche, 2020). For conventional IS research practitioners, a research study must adhere to scientific methods whereby the study can be critical reviewed based on its design, implementation and evaluation. This chapter has introduced the reader to this ideology when describing design science a "wick problem" or the different philosophical viewpoint authors may have. Considering all of this, the primary argument has been the lack of research rigour demonstrated in design science. This of course by association, extended to DSR.

To reduce the gap, Hevner et al. (2004) developed guidelines for DSR research studies. These guidelines exists to assist researchers to clearly define research boundaries while at the same time provide context for readers. It serves a template for planning, implementation, evaluating and reporting (Baskerville et al., 2018). It enables researcher to systematically evaluate an artifacts utility and usefulness in relation to the research problem, as expressed Table. 3.3

Table 3.3: Adapted DSR Guidelines (Hevner et al., 2004; Merwe, Gerber & Smuts, 2019)

#	Guideline	Description
1	Problem Relevance	The problem must pertinent and implementable to a business environment.
2	Research Rigor	The research must demonstrate a degree of planning along with the appropriate use of research methodologies to implement and evaluate research artifact.
3	Design as a Search Process	As illustrated by the DSRP (Figure 3.2), DRS requires a search strategy in order to extract kernel knowledge to satisfy laws in the problem domain.
4	Design as an Artifact	The artifact is required to demonstrate its usefulness in comparison to similar products.
5	Design Evaluation	The artifact must adhere to well-define methods to measure its utility while meticulously demonstrating its quality and efficiency.
6	Research Contributions	DSR must clearly deliver new knowledge contributions in the design artifact, design construction, and/or design evaluation in a verifiable manner.
7	Communication	DSR must effectively present its finding to a stakeholders in both the artificial and business environment.

3.4 Research Design for the Unified Query Platform

This section of chapter describes how DSR was applied to this research study on unified query platforms for NoSQL databases. As stated in section 1.5, Table 3.4 intends to provide context to the reader on how this study addressed the research questions to achieve its objectives by aligning the research questions to the data collection methods. The subsequent guidelines of the study demonstrates the research rigor to ensure the design artifact proves its effectiveness and quality.

Table 3.4: Research Questions and Data Collection

#	RQ’s	Data Collection
1	What essential guidelines must be applied when building an uniformed query platform?	Literature Review
2	What are the de facto design and architectural principles for developing an uniformed query platform?	Literature Review
3	To what extent is the uniformed query platform able to translate abstract queries to native queries for the identified NoSQL data models?	Literature Review, Experiment
4	What are the effective and efficiency determining factors that dictates how	Literature Review, Experiment

	well the unified query construct is able to translate and execute on each NoSQL database?	

3.4.1. Guideline 1 : Problem Relevance

In accordance with this guideline, academic literature on unified query platforms was extensively assessed. To ensure this study adheres to this guideline, the findings needed to demonstrate that problem is relevant to consumers and that a technological goal-orientated artifact may indeed be achieved (Hevner et al., 2004). The research problem articulated in chapter 1, section 1.3, is undoubtedly supported by the current literature on unified query solutions. The ability to interrogate and organize heterogenous data models of NoSQL storage mechanism in an uniform manner remains a challenge for stakeholders.

While strides in this field has progressed in recent years through either polyglot or multi-model native solutions, the various implementations for unified query NoSQL solutions does not fully encapsulate the four distinct NoSQL data model types as it's complex in nature. Furthermore, there is unequivocally no standard that exists today that is able to consolidate the four NoSQL data modelling types through normalize methods (Gobert, 2020; Zhang et al., 2021).

3.4.2. Guideline 2 : Research Rigor

A careful balancing act was required between research rigor and relevance for DSR studies as too much emphasis on rigor can adversely affect a studies relevance (Hevner et al., 2004). Nonetheless, rigor is important in DSR. Therefore in order to ensure that this study adheres accepted DSR practices within the IS research context. NoSQL vendor specific documentation and existing academic literature on unified query platforms imposed by the systematic literature review process were iteratively accessed throughout the study.

The prototype was constantly assessed against the research objectives defined in guideline 4 to ascertain if the requirements were met. This approach aligned perfectly with SCRUM methodologies, since it's an iterative approach to managing the software development lifecycle. This approach required the primary objectives to be translated into stories. Each one decomposed into a subset of tasks linked to requirements. The study used the V & V model, discussed in guideline 5, to perform unit and functional tests once the stories or tasks were completed, verifying if the desired goals were achieved (Olsen & Raunak, 2019). The design principles or guidelines derived from the literature and the empirical insights acquired during the research process when constructing the prototype enabled the researcher justify design and architectural choices made.

3.4.3. Guideline 3 : Design as a Search Process

Inferences made from existing literature based on ideal approaches and current shortfalls on unified query platforms guides the software development lifecycle of the prototype. The DSR is an iterative process as indicated by the DSRP model. The circumscription process repeatedly

extracts literature on recommended architectural principles and designs, influencing the prototype construction until requirements based on the research goal is satisfied.

This activity, within the scope of the research process model, involved identifying the relevant theories and frameworks used in current unified query solutions. This provided the theoretical grounding needed to informed the design and development process of the experimental prototype. The research constantly moved between gathering kernel knowledge extracted from literature and the applying the knowledge to the prototype. Hence the protype's composition was constantly refined to meet the requirements and ultimately the objectives. The conclusion of this iterative search process is reached when the requirements elicited from the research problem has been satisfied.

3.4.4. Guideline 4 : Design as an Artifact

The motivation for the study was to extract design and implementation knowledge for unified query systems that allows stakeholders to query data from multiple NoSQL sources. As previously mentioned in the guideline 2, the lack of a unified query platform that integrates to the four NoSQL category types has been discussed throughout this thesis thus far.

The envisage prototype for the unified query platform had the following principle objectives:

- Develop a custom parser that accepts a SQL-like query as input.
- Build a translating layer that accepted the parser output and generated a native query.
- Build a an executing layer that accepts the native queries as input and executes it on the supported NoSQL data stores.
- Build a reporting mechanism to determine how well the prototype operates.

This study aimed to document and demonstrate to what extent a unified query platform can be achieved. Therefore the knowledge attained through this research effort can be applied to the existing principles and practices of unified query solutions.

3.4.5. Guideline 5 : Design Evaluation

The Verification and Validation measurement model (V &V) was used to assess the prototype's effectiveness and efficiency. The evaluation method entailed conducting an experiment to simulate human-behaviour. The experimental process involved predefining a set of unified queries as input to execute varying test scenarios. The circumstances entailed variations of data retrieval, modification and insert operations. i.e. Data Manipulation Language (DML)

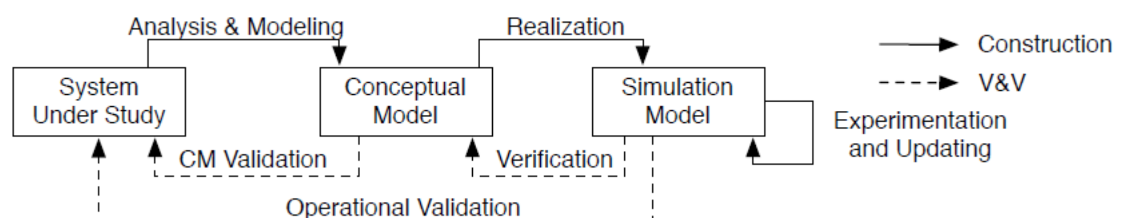


Figure. 3.3: V & V Measurement Model (Olsen, M. & Raunak, M., 2019)

Figure 3.3 demonstrates interactions between the system under study which in this context is the unified query prototype. The conceptual model embodies the kernel theories and assumptions from existing unified query solutions (Olsen & Raunak, 2019). Furthermore, it encompasses the rules and expected behaviour of the envisage solution. The prototype achieved this at various levels of granularity through adopting unit tests, integration tests, functional tests and finally user simulated testing. As the prototype construction incrementally progressed, more focus was given to simulated testing in preparation for it final evaluation run. As opposed to the unit testing which gained more focus during the inception phase of construction.

The conceptual model was operationalised into a physical implementation geared for running simulations while capturing the data of interest. Each simulated query was cloaked with the App Metric reporting module to assess its utility in tandem with the problem domain (Vaishnavi, Kuechler & Petter, 2019; Peffers et al., 2020). The study captured the following metrics:

- Apdex – measuring the performance of the prototypes unified query.
- CPU Usage – the cost in time, taken to utilise the machine's CPU.
- Memory
    - Physical – amount of RAM allocate to the query process.
    - Virtual – amount of disk memory allocate to the query process.
- Query Executions
    - Parser – time taken to parse the unified query
    - Translator – time taken to generate natives queries.
    - Executor – time taken to execute respective queries.
- Error Rates
    - Parser – error count when parsing the unified queries.
    - Translator – error count when generating native queries.
    - Executor – error count when executing the native queries.

3.4.6. Guideline 6 : Research Contributions

In its essence, this study contributes to the body of knowledge in two ways which is inherent in DSR. Firstly, the actual prototype which embodies an solution for querying multiple NoSQL storage data models via a single interface. Secondly the design and architectural knowledge attained based the empirical data from this study that can be generalized and ingrained into best practices and recommendations for unified query platforms. Therefore the contributions are as follows:

- Design and Architectural patterns applied to the prototype.
    - Approach used to translate a unified query
    - Approach used to generate and execute native queries.
- The method employed to codify the syntactic, semantic and lexical parser.
- The performance data and actual query processing output will be made public.

3.4.7. Guideline 7 : Communication

This results of this study is intended for a technical audience given the nature of the research study. The study produced an technological-orientated prototype used to measure the degree of how a unified query platform. The report was generated from data collected for this study was synthesised and presented the systematic investigative findings. It draws a conclusion indicating how the research goal was accomplished.

3.5 Summary

This chapter presented the research methods applied this study and how the systematic research enquiry was conducted to achieve its objectives. The chapter firstly describes the philosophical stance the study adheres to. It provides clarity on how this study perceived data within its philosophical context. The research endeavour makes use of Design Science Research as a strategy. It justifies why it used and how it fits within the research paradigm.

Using DSR as a mode of enquiry, the study adheres to the guidelines proposed by Hevner et al. (2004). To ensure the research conforms to the necessary rigor, often criticized in DSR undertakings, it follows the Research Design Process Model proposed by Vaishnavi, Kuechler & Petter (2019). The process model safeguards against transgressions demarcating clear boundaries of each activity in the research cycle.

CHAPTER FOUR : UNIFIED QUERY PLATFORM DESIGN AND IMPLEMENTATION

4.1 Introduction

This chapter focuses on the design and implementation of the unified query platform artifact outlined in section 3.4. The creation of the artifact is motivated by the research problem identified in chapter 1, section 1.3. Guideline 1, section 3.4.1, expanded on the problem describing the relevance this study has in the current environment.

In chapter 2, the reader was presented with the systematic literature review to emphasise the current approaches, accepted de facto standards and the shortcomings of current unified query systems. This is indicative of the two research questions addressed (i.e. *RQ1* - *RO1*; *RQ2* - *RO2*), whereby the reader's attention was drawn mostly to the theoretical design and architectural aspects of unified query solutions.

Chapter 3 provided the reader with the research recipe on how the objectives of this study was achieved. It set out the DSR guidelines the study adhered to, outlined in section 3.4. This provided the foundation to transition to the third research sub question in Table. 4.1.

Table 4.1: Research Questions and Objectives

#	RQ's	RO's
3	To what extent is the uniformed query platform able to translate abstract queries to native queries for the identified NoSQL data models?	To design and implement a unified query construct as middleware that collectively transforms, routes and executes an abstract query to each native NoSQL data models.

4.2 System Design Goals

The goal of this prototype was to provide a high-level unified query construct that is database-agnostic; capable of querying data across the four types of NoSQL storage models (Kolonko & Müllenbach, 2020). The query language must offer a consistent syntax and a set of operations that can be used to express queries in a generic manner. The design objectives to support the aim of this research study was steered by Hevner et al. (2004) DSR guidelines.

Mardiana, 2020 states DSR is utilised, "when a researcher needs to create something (artefacts, e.g. software, hardware, process) in order to solve the problem in organization or changes the society toward the better, while at the same time learning and accumulating knowledge during the process."

Based in the literature reviewed as directed in guideline 3 of the DSR strategy, several requirements were identified (Table. 4.2). Each requirement was linked to a component responsible for a specific functionality in realising a unified query platform. These components are like a "spoke in a wheel", thus dependant on each other to achieve the design objectives. The component related to the requirement as follows:

- R1 - Repository Metamodel
- R2 - Query Language Construct
- R3 - Query Processing Engine
- R4 - Query Executor

Table 4.2: Artifact: Design Requirements

#	Requirements\Stories	Tasks
DR1	Create a 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.
		Link global schema to native schemas.
DR2	Create query parser for the unified query.	Build a lexer for input characters.
		Build a query syntax tree.
		Build a semantic engine.
DR3	Create a query translator.	Build Syntax and Semantic Matching engine.
		Build Feature Mapping engine.
		Build Query Optimization engine.
DR4	Create query executor	Build a database adapter for NoSQL databases.
		Map native results to a global view.
DR5	Create a logger.	Integrate App Metrics into unified query system.

4.3 System Overview

The overall system architecture of the unified query platform is depicted in the conceptual framework providing a high level view of the various components based on the specified requirements, DR1 to DR5. The conceptual framework served as the starting point since it's independent of the underlying database technology (Hewasinghage et al., 2021; Atzeni et al, 2020; Candel, Ruiz & García-Molina, 2022). Essentially, the scope of this study was bounded by the polyglot system shown conceptual framework illustrating how a unified query model for multiple variants of NoSQL database technologies may be achieved.

4.2.1 Conceptual Framework

The conceptual framework in Figure 4.1 illustrates the rudimentary path a query follows when navigating through the components which makes up the unified query platform. As the query transverse through the components the interactions are audited at each point thus providing transparency to its inner workings. For the sake of completeness and better understanding to the reader; the entire query platform is grouped into four categories namely query commands, query results, data streams and monitors.

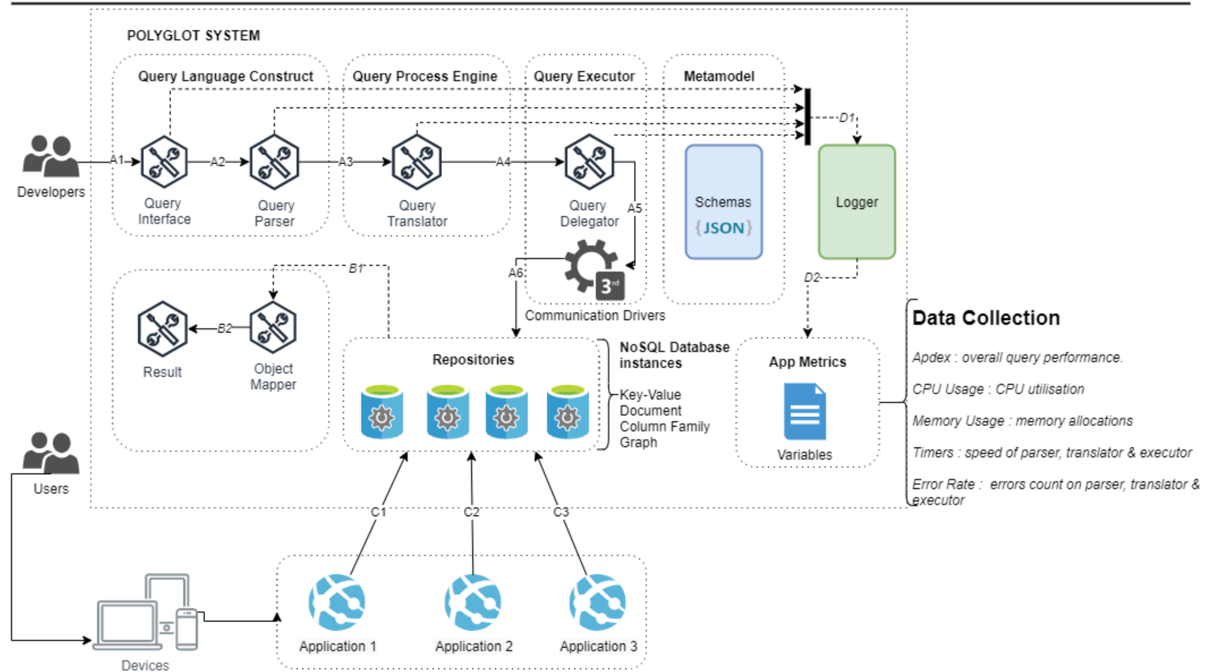


Figure. 4.1: Unified Query Conceptual Framework

- Query Commands
  - A1 - Represents the query request, Fetch, Modify or Add.
  - A2 - Validates query against the unified parser.
  - A3 - Translates the abstract query into natives queries.
  - A4 - Directs native to the respective NoSQL database instances.
  - A5 - Connects to the targeted NoSQL instance.
  - A6 - Sends query request for execution.
- Query Output
  - B1 - Represents the query output.
  - B2 - Binds query output to an object representing the data queried.
- Data Stream
  - C1,C2,C3 - Novel applications the feeds the repository with data.
- Logs
  - D1 - Determines which metrics to log.
  - D2 - Logs the performance metrics.

4.2.2 Applied Abstraction to the Prototype

As noted by a Ramadhan et al. (2020), constructs of unified query solutions are primarily abstractions of mathematical formulas to encapsulate heterogenous data models coupled with an algorithms to generate abstract syntax trees and dictate execution logic. In fact this study is based on the following mathematical abstractions whereby *q(n)* represents the native or targeted query for each instance category of a NoSQL database. *DS* represents the data source which consolidates the four types data storage models. i.e., *GR* - Graph, *KV* - Key-Value, *DO* – Document-Orientated, *CO* - Column- Orientated data stores.

The data source is represented by $DS \rightarrow GR \cup KV \cup DO \cup CO$, indicating which the NoSQL data storage models are supported. The query parser ensures the unified query conforms to the signature, $S_{lss} = \sum_{i=0}^{n-1} k^i, k < (lexical[i] \wedge semantic[i] \wedge syntax[i,])$, whereby the unified is required to prove it conforms to the lexical, semantic and syntactic rules of the unified platform. The query translator verifies if the targeted data model, *dm(k)*, specified in the unified query is an element of the data source:

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

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):

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

The query executor subsequently directs t(k) to appropriate NoSQL database instance to be executed. If $q(n) = \prod_{k=1}^{DS_n} k, \exists_n[\emptyset, n]. t(k). dm(k)\ holds$, the native query successfully executed on the target storage model. Finally, the object mapper wraps the output of each target query into a result, $r_i = o \in [q(0), \dots q(n)]. (k \geq q(k))$.

4.4 System Design

This section details the core features of the prototype for the unified query platform. In addition, it also aims to rationalise and justify on certain choices that were made during the design and construction phase underpinned by the RSDP Model in section 3.3.1. The design encompasses a collection of underlying theoretical concepts discussed in the literature review.

4.4.1. DR1 : Metamodel Repository

The metamodel is one of the most critical components of any unified query system. Its primary is purpose was to serve as an intermediatory between the unified and the native schemas (Kolonko & Müllenbach, 2020, Hewasinghage et al., 2021; Glake et al., 2022). The prototype required a meta model repository cataloguing each the storage mechanisms schematics, data types, and indexes (Appendix E). The metadata assists in the prototype's query parsing mechanism, performing basic validations to ensure that the specified fields are support by the unified query data model.

It aids the query translator resolving native references at runtime and assist in generating the appropriate native query constructs. It informs the query processing engine the optimal query to create by inspecting the relevant native storage mechanism schematic information such as indexes and unique keys.

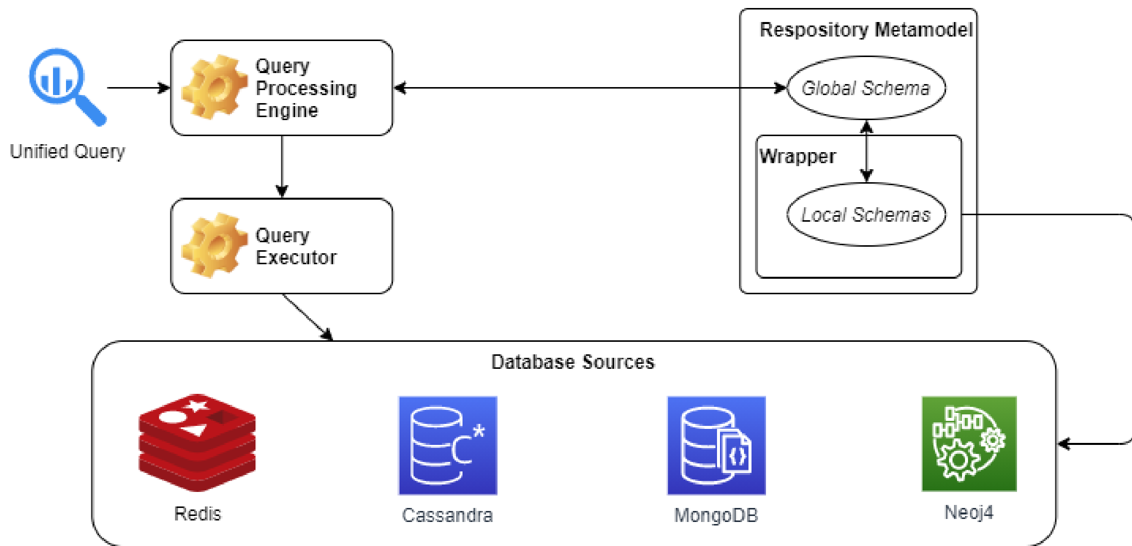


Figure. 4.2: Prototype's Metamodel

Figure 4.2 aims to illustrates a high-level interaction view of the prototype's repository metamodel in relation to the other components within the polyglot resolution. The design choice for the metamodel approach is that of a Global-as-View described in section 2.4.6. The global schema for the prototype is expressed as a function of the local NoSQL databases (Endris, 2020). The GAV mediator acts as an intermediary between the unified query language and the underlying NoSQL databases, abstracting the intricacies of data source integration.

The Global-as-View serves the prototype in two ways. Firstly, its ensures that the attribute and model references in unified query actually exists natively. Secondly, it guides the native query construction process by analysing their relationships, as well as constraints and modelling rules but not the concrete syntax of the language.

4.4.2. DR2 : Query Parser

A core feature of the query language construct for the prototype is the AST for the unified query platform. The AST encapsulates the syntactic hierarchical structure of the custom designed abstract query construct, denoting the essential mechanisms and relationships between various elements in the query, such as keywords, operators, identifiers, literals, and expressions (Zhang, 2020; Yang, Zhang & Tong, 2022). The AST abstracts away the specific syntax of different vendor specific query mechanisms and provides a common structure that facilitates query construction and processing.

Structure of the Prototype's Abstract Syntax Tree Structure

The prototype syntax supports three query statements, each adhering to a specific structure. The AST consists of the FETCH statement representing the retrieval of data, the ADD statement for inserting data and the MODIFY statement for updating data. The statements are comprise of nodes connected through a parent-child relationship forming a tree-like hierarchical structure as illustrated in Figure 4.3, 4.4 and 4.5. The root node represents the overall query, while the child nodes represent the constituent parts of the query.

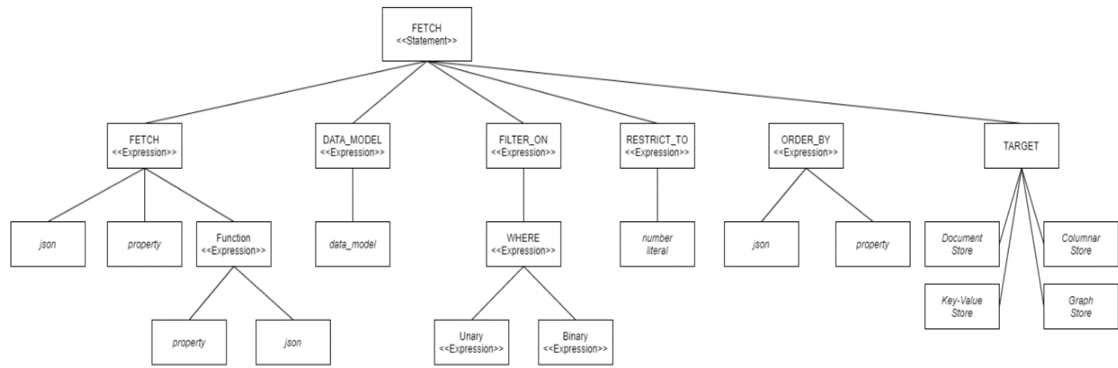


Figure. 4.3: AST - Fetch

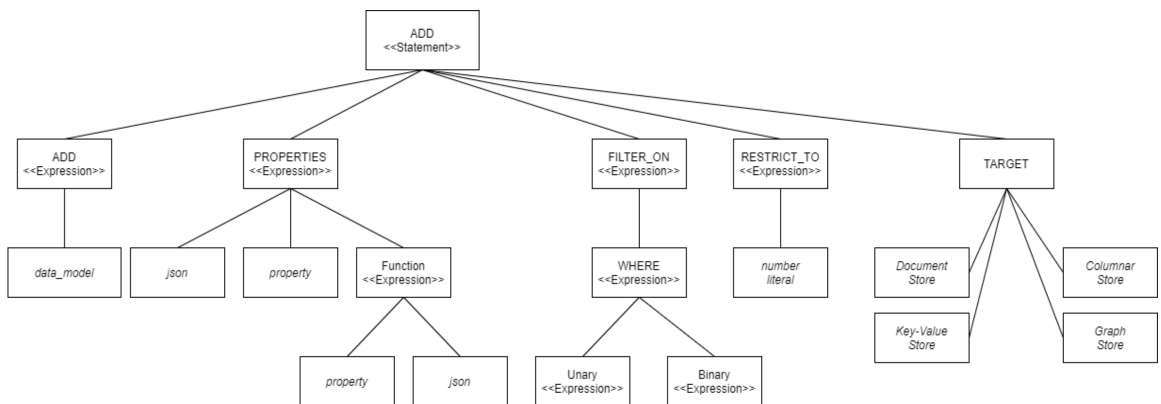


Figure. 4.4: AST - Add

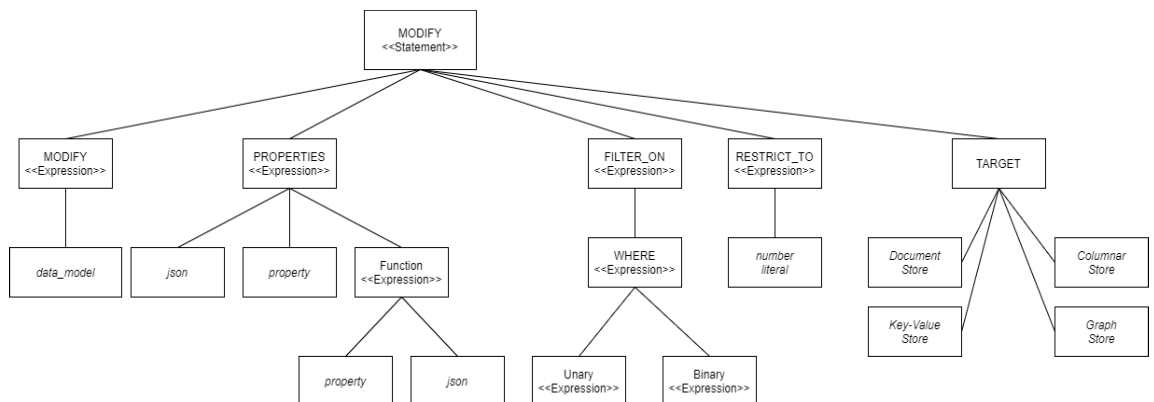


Figure. 4.5: AST - Modify

Abstract Syntax Tree

The first step in the design process of the protype's AST was to determine how context and meaning can be given to the query language (Duracik et al., 2020; Zhang, 2020). Part of the design phase was determining what the basic units of the language ought to be, ultimately producing in the abstract or unified query. This was achieve by arranging elements of the query into an organised structure as in shown in Appendix F. The decomposition of the query elements forms the lexical analysis phase whereby tokens are generated, i.e. the basic units of the query language.

To achieve this, the prototype employed a lightweight library called Superpower that facilitates the construction of token-driven parsers embedded directly in the source code (Blumhardt, 2022). 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" (Blumhardt, 2021). A demonstration of the lexical activity in Appendix G reveals how the tokens are generated by the prototype as per a given input.

The tokenization of the query input string involved recognizing keywords, identifiers, literals, and operators. Hence the prototype has a lexer feature embedded in the query parser component which scans the text; spawning a stream of tokens that serves as input for the subsequent parsing phase (Appendix H). The parsing phase examines the stream of tokens generated by the lexer, systematically building the AST based on the grammar rules of the unified query language.

The prototypes applies a parser combinator technique whereby multiple parsers are accepted as input to create a new parser as output. In computer science (Öztürel, 2022), this approach embodies the mathematical concept called a Higher-Order function (HOF). HOF states that a function must at least take one or more functions as an argument which in turn yields as a function result. The parser combinator enables the protype to modularise the sections of the query language using the demarcating locations from the token stream by recursively traversing through the text. The demarcated locations assists the program indicating where the parser should start and stop. This recursive decent strategy, follows a top-down procedure inspecting the terminal and non-terminal symbols based on the syntactic rules governing the grammar of the unified query; resulting in grouping a disjointed set of nodes (Guo et al., 2020).

The semantic analysis performed by the prototype extracts the meaning from the text in preparation for the actual invocation on the unified query. The pattern recognition of links between the keywords, identifiers, literals, and operators validates the AST's to ensure its correctness and consistency. In summation, while the syntax feature ensured the input is well-formed; the protype's semantic feature determines if the intent of the query is aligned to the correct action of the native data store. Furthermore, it provides supplementary annotations should the input transgress any semantic rules, i.e. parser errors.

4.4.3. DR3 : Query Translator

The prototype's unified query expression stated in a high-level language encapsulated by the AST, which is responsible for inspecting the query intent and translates into native queries that can be executed by the targeted data store. The translation process required several features to come to fruition for the processing engine to be able to successfully create the native query language:

- Syntax and Semantics Matching
- Feature Mapping

- Query Optimization

Syntax and Semantic Matching

A unified query intended to target multiple types of databases, will innately have different syntax and semantics compared to the native query languages (Candel, Ruiz & García-Molina, 2022). The query processing engine requires the ability to find the equivalent meaning and grammar in order to successfully execute converted queries. Therefore, the semantics of the unified query must map to the corresponding semantics in the native query language of the target NoSQL database.

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 transforming the syntax of a unified query into the specific syntax supported by the native query language of the target NoSQL database. Rewriting the unified query's expressions, keywords, identifiers, literals and operators to match the syntax of the native query language safeguards and ensures adherence (Koutroumanis et al, 2021).

Feature Mapping

The high-level query language created for this prototype 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 the cases where it is unable to, it defaults to a rudimentary intent of the native query. In general, features for database management systems are naturally influence by the applicable use cases (Davoudian, Chen & Liu, 2018; Oussous et al., 2018). 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 4.3. Therefore the prototype requires an additional abstraction layer for the Redis data store to circumvent this issue which it currently does not support.

Table 4.3: 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 processing engine maps these features to appropriate native constructs or techniques in the native query language, ensuring the preservation of the expected functionality. Since the numerous NoSQL data stores for the prototype, as covered in section 1.9, aims to provide a single query interface; the approach to decoding the individual native queries depends on the degree of complexity of the supported models. Specialized strategies for each of the inherent data stores were built. Thus establishing clear boundaries between the various NoSQL translation layers. The design approaches are further discussed in section 4.5.

Query Optimization

The query optimizer plays an key role in the efficiency of a unified query language for the NoSQL 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 (Khan et al., 2019;). As a consequence, it aims to shift the I/O, memory and CPU processing power to the respective DBMS reduces 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 (Khine & Wang, 2019).

Once the query has parsed the unified construct rules, the next step is to rewrite the query expression into an equivalent, more efficient form. To do this, instructions are extracted from the unified query language while unpacking the metadata from the metamodel (section 4.4.2) to find the suitable native tables, collections, nodes with the corresponding attributes/relationships. The query optimizer relies heavily on the metamodel to produce a native query in the most effective form.

4.4.4. DR4 : Query Executor

The query executor is responsible for natively running queries produced by the query processing engine against the respective NoSQL data sources. It takes care of establishing the database connections, the authentication procedures and data transfer between the unified query platform and the data source, similar approaches to BigDawg, NoDA (Koutroumanis et al., 2021; Zhang et al., 2021). 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. For clarification sake, it does not merge the different data sources but rather ensure that each one conforms to the unified data model. Any errors and exceptions that may occur during query execution process provides the appropriate error messages back to the query interface.

4.4.5. DR5 : Metrics Logger

The metrics used in this the experiment was programmatically embedded into to prototype. The protype utilised is an open source library called App Metrics to measure various performance aspects of components that makes up the unified query solution (app-metrics.io, 2021). The report modules provided a set of libraries whereby unified query parser, the translator and the executor could be scoped.

4.5 System Construction

At this point of the chapter, the researcher delves into how the system design requirements created in the previous section were tied together which eventually produced a testable instantiation. Although these aspects were not explicitly expressed in section 4.2, due to the very nature of this study, it is an important non-functional requirement. The prototype for the unified query platform applied various design patterns to address different aspects of functionality within its architecture.

While these design patterns have existed for almost thirty years at this present moment, it's still relevant today and has become the corner stone of solve object orientated programming problems. The design patterns used in this study are classified as a behavioural design patterns (Gahlyan & Singh, 2018). As the prototypes main concern was how objects interface with one another in an efficient way by finding common interfacing patterns. It enhanced the modularity, extensibility, scalability and maintainability of the unified query platform (Wedyan & Abufakher, 2020).

4.5.1. Query Intent

An intrinsic functionality of the prototype was determining the intent of the unified query to produce the expected result. The chain of responsibility design pattern was chosen, where at runtime the prototype decides which command to execute (El Maghawry & Dawood, 2010). The prototypes defines *Fetch*, *Add* and *Modify* commands as handlers each responsible for its own interpretation of the request. It shares a common interface which is responsible for dispatching client query request to the appropriate command handler depending on the data enquiry (Gahlyan & Singh, 2018). The command handlers contains the query parser and translator logic discussed in section 4.4.2 and 4.4.4.

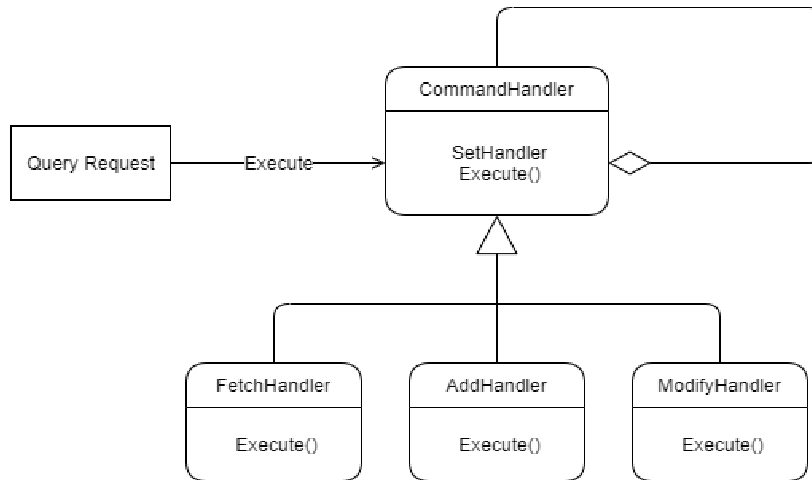


Figure. 4.6. Chain Of Responsibility Pattern : Prototype Commands

Figure 4.6 provides an illustrative view on how the pattern is applied to the protype. This pattern has been widely used in cases whereby system messages dictates the execution result (Wedyan, F. & Abufakher, S., 2020). New instances of each command type are create as the program starts up resulting in a chain of objects. To improve the efficiency of the executing processing chain of objects; the collection of concrete handlers, i.e. commander handlers , was set up as a dictionary where the command type is the unique key. The query request passed to handlers is tagged with the appropriate command type which is used to find the corresponding handler in the chain to execute (Table 4.4). When the command is not found in the dictionary, this prevents an action from being taken, unified query request is aborted with an error message.

Table 4.4: Pseudocode : Query Intent

$Q \rightarrow a\ query\ intent$	// represents a query intent
$N \rightarrow \{n_1, \dots, n_n\}$	// AST := Terminal and Non-Terminal nodes
$f \rightarrow fetchhandler(N)$	// fetch command
$m \rightarrow modifyhandler(N)$	// modify command
$a \rightarrow addhandler(N)$	// add command
$cmd \rightarrow \{ f \mid m \mid a\}$	// command handlers
$cmd \subseteq Q$	// command handlers is a subset of a query
$:= \boldsymbol{QueryIntent(q)}$	// if *q* represent a query
$\boldsymbol{if}\ q \in Q\ \boldsymbol{do}$	
$\quad \boldsymbol{if}\ q.cmd \in f\ \boldsymbol{do}$	// loop through each command type
$\quad\quad f(N)$	// invoke *fetch* command
$\quad \boldsymbol{else\ if}\ q.cmd \in m\ \boldsymbol{do}$	
$\quad\quad m(N)$	// invoke *modify* command
$\quad \boldsymbol{else\ if}\ q.cmd \in a\ \boldsymbol{do}$	
$\quad\quad a(N)$	// invoke *add* command
$\quad \boldsymbol{else}$	
$\quad\quad InvokeError$	//return error message

4.5.2. Query Path

In the next non-functional requirement, the prototype needed to ensure the correct native query path is taken based on the unified query intent. A strategy pattern was applied which selected and ensured the correct algorithm was enforced based on query elements specified in target clause within the AST. Each of the supported NoSQL data storage models were defined as descendants as part of a family algorithms shared by the same ancestor (El Maghawry & Dawood, 2010).

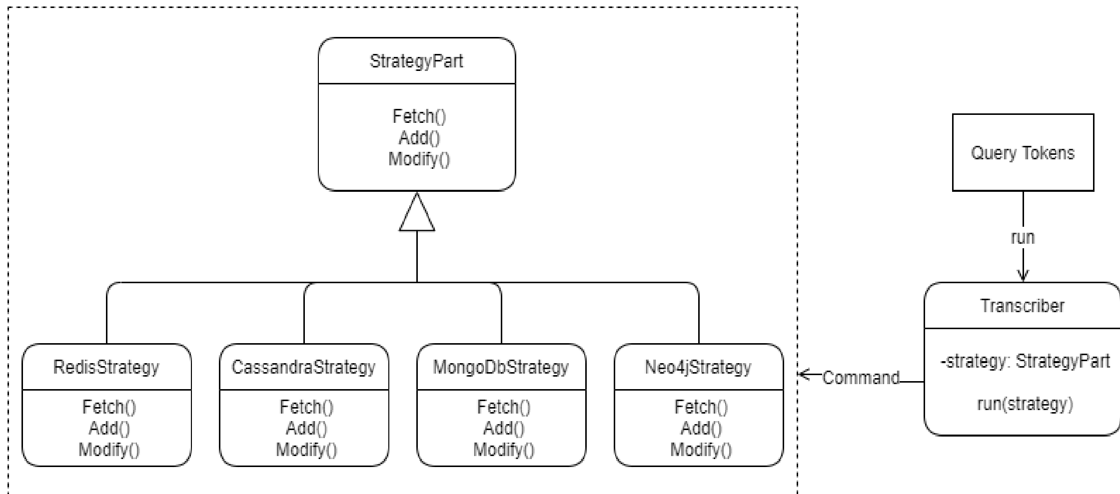


Figure. 4.7: Strategy Pattern : Query Generator and Executor

Each of the supported NoSQL data models are deemed as specialised classes in the prototype, liable for building a collection of visitors, described in section 4.5.3, to be executed by the query generator. The prototype receives the query intent as input; matches the command and storage target to the relevant strategy (Table 4.5). In preparation for the native query generators, during the translation process the repository metamodel is used to find the unified field's equivalent native field contained in the AST. If no matches are found then the field will be excluded. The prototype consciously builds a collection of class instantiations represented as visitors to closely resemble the structure of native query languages it needs to create. Finally, once the native queries has been created by the query generator, the strategy pattern send the output back to the calling method for execution.

Table 4.5: Pseudocode : Query Path

$rs \rightarrow redis$	// redis strategy
$ms \rightarrow mongdb$	// mongodb strategy
$cs \rightarrow cassandra$	// cassandra strategy
$ns \rightarrow neo4j$	// neo4j strategy
$T \rightarrow transcriber\backslash convert$	// represents the conversion process
$SP \rightarrow \{sp_1, \dots, sp_n\}$	// represents collection of query path strategies
$sp_n \in (rs \mid ms \mid cs \mid ns)$	// each native strategy is an member the query path strategy
$sp_n \ni T$	// represent a member of the transcriber
$DS \rightarrow (GR \mid KV \mid DO \mid CO)$	// prototype data sources

$q \rightarrow Query$	// represent a query
$n \rightarrow Native\ Query$	// represents a generated native query
$DS \subseteq Q$	// data source is a subset of the query
$I \rightarrow QueryIntent(q)$	// query intent function (see 4.5.1)
$\coloneqq \boldsymbol{QueryPath(q)}$	// loop through each strategy members
$\boldsymbol{if}\ q \in I\ \boldsymbol{do}$	
$\boldsymbol{if}\ e \rightarrow \exists(q.cmd)\ \boldsymbol{do}$	// determine if command is valid
$target\ storage \rightarrow q.DS$	// set targeted storage
$\boldsymbol{for}\ each\ sp\ \in SP(target\ storage)\ \boldsymbol{do}$	// match strategy to targeted storage
$\quad \boldsymbol{if}\ sp \subseteq KV\ \boldsymbol{do}$	
$\quad\quad n \rightarrow T.Run(rs)$	// invoke *redis* strategy
$\quad \boldsymbol{if}\ sp \subseteq CO\ \boldsymbol{do}$	
$\quad\quad n \rightarrow T.Run(cs)$	// invoke *cassandra* strategy
$\quad \boldsymbol{if}\ sp \subseteq DO\ \boldsymbol{do}$	
$\quad\quad n \rightarrow T.Run(ms)$	// invoke *mongodb* strategy
$\quad \boldsymbol{if}\ sp \subseteq GR\ \boldsymbol{do}$	
$\quad\quad n \rightarrow T.Run(ns)$	// invoke *neo4j* strategy
$return\ \boldsymbol{n}$	

4.5.3. Query Generator

The visitor pattern was harnessed to generate the native NoSQL queries for the prototype. It is triggered by the query translator component which is discussed in section 4.5.2. The native query elements are represent as “Visitors” which directly correlates to elements of the tokens generated by the query parser. This pattern is quite powerful as it allows a class instantiation to add functionality without changing the structure of the class, making it scalable (El Maghawry & Dawood, 2010).

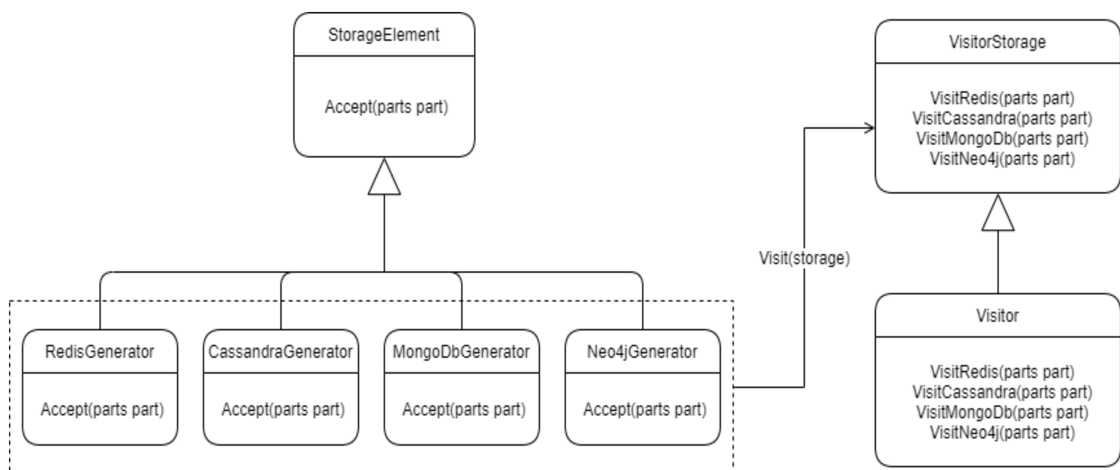


Figure. 4.8: Visitor Pattern for NoSQL Code Generators

The supported NoSQL data storage models in the scope of this study each has its own unique code generating implementation, shown in Figure 4.8. It decouples the processing logic from

query components thus isolating query elements enabling new features to be added without affecting existing parts. This pattern enables the prototype to traverse through the different elements of the query expressions, building parts of the native query while maintaining its internal state, i.e. the 'whole part' or native query (Table 4.6). As the prototype navigates through the organised parts, it invokes other visitors hence complex query structures able to be built in a systematic and controlled manner.

Table 4.6: Pseudocode : Query Generator

$rg \rightarrow redis$	// redis code generator
$mg \rightarrow mongdb$	// mongodb code generator
$cg \rightarrow cassandra$	// cassandra code generator
$ng \rightarrow neo4j$	// neo4j code generator
$VS \rightarrow \{vs_1, \dots, vs_n\}$	// collection of visitors, i.e. query elements
$G \rightarrow (rg \mid mg \mid cg \mid ng)$	
$G \subseteq vs_n$	// each storage generator has a subset of visitors
$SE \rightarrow \{qe_1, \dots, qe_n\}$	// parts the supported storage query elements
$I \rightarrow QueryPath(q \rightarrow Query)$	// query path function (see 4.5.2)
$\coloneqq \boldsymbol{QueryGenerator(i)}$	
$\boldsymbol{if}\ i \vdash I\ \boldsymbol{do}$	// determine if query path has been established
$strategy\ path \rightarrow i.DS$	// set strategy path based on specified data source
$VS \rightarrow BuildVisitors(i.query_elements)$	// build visitor parts
$\boldsymbol{for}\ each\ part\ in\ VS\ \boldsymbol{do}$	// loop through parts
$\boldsymbol{if}\ part \in rg\ \boldsymbol{do}$	
$rg.Accept(part)$	// build native *redis* parts
$\boldsymbol{if}\ part \in cg\ \boldsymbol{do}$	
$cg.Accept(part)$	// build native *cassandra* parts
$\boldsymbol{if}\ part \in mg\ \boldsymbol{do}$	
$mg.Accept(part)$	// build native *mongodb* parts
$\boldsymbol{if}\ part \in ng\ \boldsymbol{do}$	
$ng.Accept(part)$	// build native *neo4j* parts

4.6 System Review

At the start of the prototype construction process, this study had to ascertain that the requirements specified in section 4.3 were satisfied. In this phase of the research process as outlined in section 3.4.5, illustrated by Figure. 4.9; the three stages of the V & V measurement model were satisfied. To ensure the objectives were met during the development phase, unit tests were created for each critical component to validate and verify the expected output and minimize any software bugs. A total of 179 units were created, safeguarding all the fundamental test scenarios.

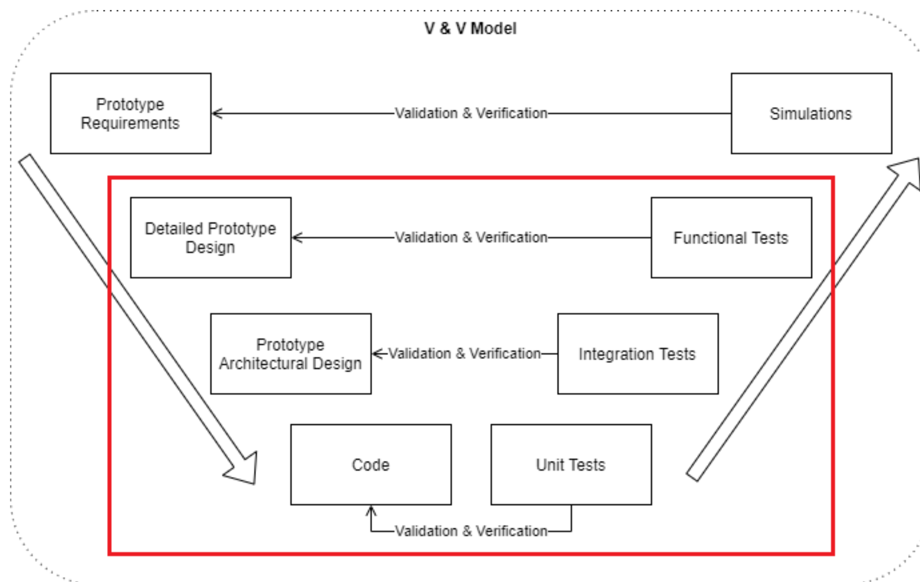


Figure. 4.9: Verification & Validation Measurement model

The unit tests were followed by integration tests to confirm that the components are able to communicate with each other. This included establishing database connections, loading libraries to determine compatibility and how well programming modules work together given varying inputs. The functional tests conducted determined whether the prototype features operates as per the requirements. The verification process checked that the prototype was developed in the right way according to the defined specifications. It substantiated whether the developed artifact fulfilled the requirements. The validation process checked and compared that the actual versus the expected output. Any deviation from the expected output was deemed a failure and subject to the source code being refined through multiple iterations until it achieved the desired result. The question asked throughout this process, '*is the correct product being built ?*'.

4.7 Summary

This chapter delved into the design and implementation process of the unified query platform. It lists the design requirements stating how it aligned to the aims and objectives of this research study. A conceptual framework was illustrated to the reader to visually articulate and clearly demonstrate the system boundaries of each component that makes up the unified query prototype. Throughout this chapter the researchers justified certain design and architectural decisions made during the development of the prototype.

It makes a distinction between the functional and non-functional design requirements of the prototype. The functional requirement comprised of components shown in the conceptual framework, i.e. the parse, translator, executor and repository model. The non-functional requirements dealt how the research employed object orientated design programming patterns to efficiently mapped the functional to the non-functional requirements. This demonstrates how the prototype bridged the gap between the unified query and native queries.

CHAPTER FIVE : PROTOTYPE EVALUATION AND RESULTS

5.1 Introduction

In chapter 4, the design and implementation for the unified query platform was discussed, describing key features for each of the components. Chapter 4 covered how the physical implementation was attain through varying design and architectural approaches. This chapter evaluates the validity of the prototype by examining the results from the simulations conducted. The prototype was subjected to varying test scenarios whereby each query output and performance metrics were collected over a time series.

Table 5.1: Research Questions and Objectives

#	RQ's	RO's
4	How effective and efficient is the performance of the applied query processing strategies in the unified query platform?	To determine the effectiveness and efficiency of the unified query construct considering data integration, query execution, and result retrieval.

The study evaluates and synthesises the results of a unified query platform in accordance to its effectiveness and efficiency with respect to the problem domain. The chapter begins by drawing the reader's attention to the problem domain of for this undertaking then proceeds to the research objectives as shown in Table 5.1 as well as the methods used to gather data and document the results.

5.2 The application of DSR to the problem domain

The iterative nature of the DSR process aligned with the technological goals of prototype in the design and implementation process. The existing body of knowledge as it relates to unified query platforms provided the scientific and technical grounds to refined the prototype and observe outputs in a meaningful way (Vaishnavi, Kuechler & Petter, 2019). Since this research study adheres to the guidelines of Hevner et al., 2004, the objective for this study is "to grow the prescriptive knowledge base of purposefully designed artifacts to improve human capabilities".

5.3 Experimental Overview

To achieve the aim and objectives of the study as articulated in Chapter 1, section 1.4; an experiment was conducted to answer the associated sub questions posed. The prototype personifies the objectives of the study inspired by the research questions. The objective of this experiment is to evaluate the efficiency and effectiveness for the unified query platform prototype as discussed in Chapter 3, section 3.4.5. The developed prototype and simulations were conducted under system conditions conducive to meet the technological goals.

The unified query platform was designed to retrieve data from one or more of the targeted native storage options. As illustrated in Figure. 5.1, this experiment requires no human participants due to the technical nature of the research study. The research conducted was more focused on the architectural makeup of the prototype's unified query platform.

The experiment was controlled through a predefined set of queries which forms part of the test automation process which is discussed in further detail in section 5.4.5. Considerations were also made in terms of the machine's hardware capabilities and software requirements to realise the envisage solution.

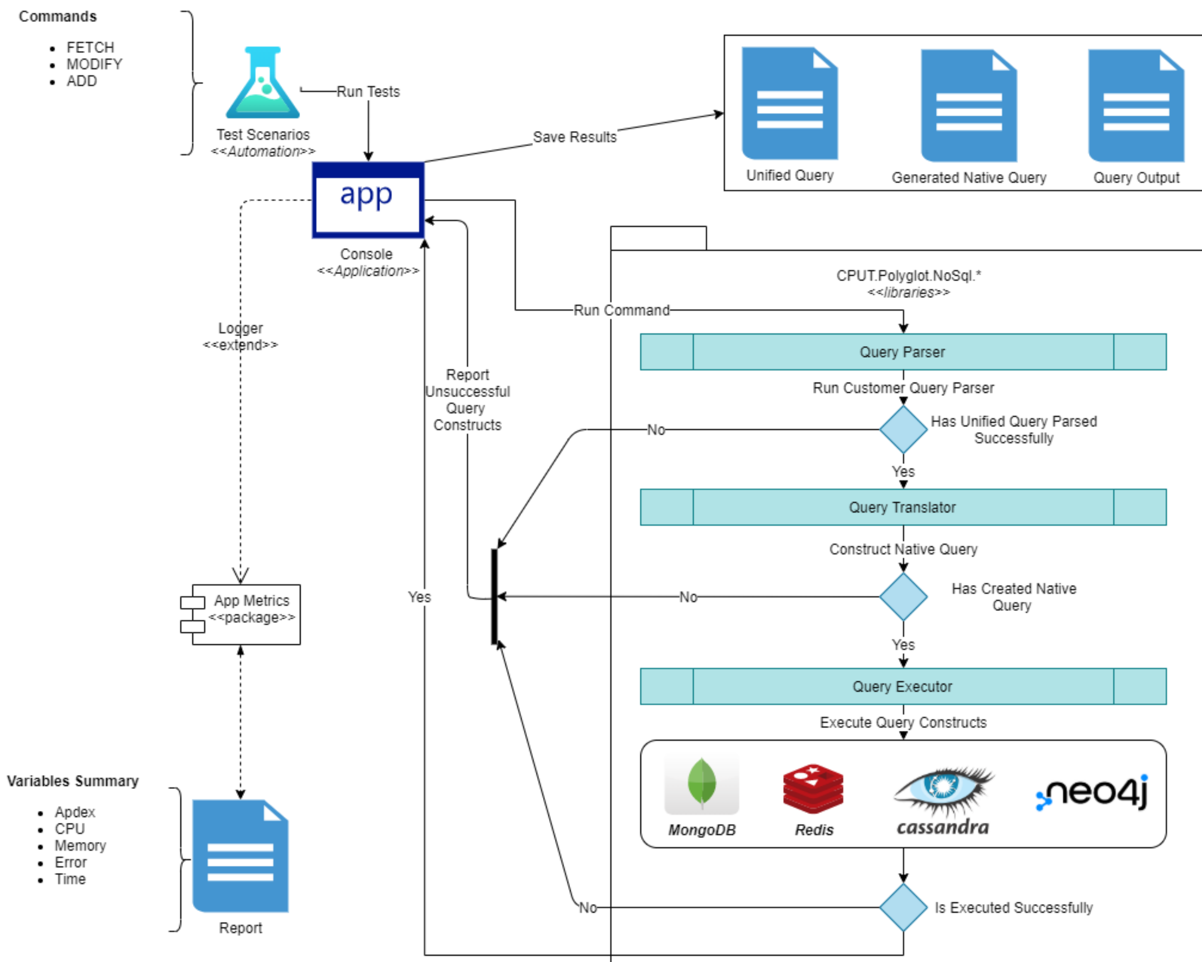


Figure. 5.1: Experimental Overview

5.3.1. Participants

The participants in this study was a subsystem within the prototype. The subsystem builds a collection of executable unified queries and delegates it to the entry point of the unified platform system. It is responsible for simulating human user interactions and plays a critical role in evaluating how well the prototypes operates (Olsen & Raunak, 2019). The participants subjected the unified query platform to various scenarios to emulate human behaviour. This consisted of defining data retrieval, modification and data insert query commands. The deterministic result which participants invoked was encapsulated as output files which highlighted performance and the correctness of the prototype.

5.3.2. Procedure

An key step in the experiment was to assign tasks to participants. The tasks captured behaviour and intention of the unified query. Each task was uniquely labelled, for the purposes of identifying metrics to the associated participant. The intention of the tasks was to set goal for the participants. The behaviour on the other hand, contained the characteristics of the unified query. This involved query types such a FETCH, MODIFY and ADD commands; the level of complexity which directly correlates to the intricacies of generating a native query depending which NoSQL data stores were targeted.

As part of the procedural process, the research had to ensure the experiment goals were aligned with the research objectives. Therefore the individual of participant's represents a subset of research objectives. This enabled the researcher to assess to what degree a query goal was reached. Subsequently, the researcher could judge whether or not the intention of the query match the actual behaviour and the overall utility of the of the prototype.

5.3.3. Hardware

The prototype simulations performed were conducted on a notebook with following specifications:

Table 5.2: Notebook Specifications

Specifications	
Processor	Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz 2.30 GHz
Installed RAM	16,0 GB (15,6 GB usable)
System type	64-bit operating system, x64-based processor
Cores	4

5.3.4. Software

The prototype was developed within the context of the following software environmental arrangement as indicated by Table. 5.3 and Table. 5.4.

Table 5.3: Development system

#	Application	Description
1	IDE	Microsoft Visual Studio 2022 Community Edition
2	Programming Language	C#
3	Framework	.Net Core 6 (platform independent)
4	Reporting	App Metrics

Table 5.4: Database system

#	Database	Version
1	Redis	3.0.504
2	Apaches Cassandra	3.11.13
3	MongoDB	6.0.0
4	Neo4j	4.4.5

5.3.5. Ethical Considerations

There were no human participants in this study, therefore ethical aspects such as data privacy and security for the working datasets does not apply. Section 5.4.2 elaborates on how the datasets were gathered. However, the research still had to comply with any relevant guidelines and regulations. This related to the use of the development and distributed software tools used to develop and evaluate the unified query prototype. Firstly, the researcher specifically sought out distributed software that is governed by open source technology principles. Secondly, the software that requires a paid subscription such as the Microsoft Visual Studio IDE, does permit it use on condition for academic purposes. Finally, careful consideration has thus been given to guiding principles like fair distribution and use, compliance standards and regulations; and respect for intellectual property. Throughout this experiment the use of distributed software and tools are also mentioned to inform and to enable the reader to distinguish what has been intellectual property has been incorporated.

5.4 Experimental Setup

This section provides a detailed description of how the experiment was created to evaluate the unified query platform. It outlines the activities executed such as the how the database models were created, how data was sourced and generated, the participants involved and the metrics gathered during the experimental process.

5.4.1. Data Models

Before embarking on the experimental process, a student database was created for each of the targeted NoSQL data stores in scope for the research study (Appendix A, B, C and D). The purpose of the database was to construct a real-world scenario containing student information, shown in Table. 5.4. The models were necessary beforehand, as it directly served as input to the metamodel and the query processing engine discussed in Chapter 4, section 4.4.3 and 4.4.2 respectively.

Table 5.5: Student NoSQL Repositories

#	Database	Ideal
1	Redis	A student profile the manages user online sessions.
2	Apache Cassandra	Serves as a repository for student information.
3	MongoDB	Contains student registration information
4	Neo4j	Contains student transcript information

While the varying data models exists in different NoSQL databases, the impetus was to create models that shared common data reflecting similar concepts. Even though the respective databases shared common data concepts, the properties in most instances had different naming conventions and more importantly different schema structures. Consequently, requiring the metamodel to serve as a bridge to the native storage databases (Glake et al., 2022). Once the respective storage models were created, an additional data model layer represent as a file was required to encapsulate all of the native storage option for the prototype in a single view, i.e. - a Global-as-View.

5.4.2. Data Generation

The data generation process was gathered in two phases. The first phase was obtaining student data for the experiment from an online source (onlinedatagenerator.com, 2023). The latter phase supplemented the initial dataset by computing additional data via the prototype itself. The initial dataset consisted of the two files in JavaScript Object Notation (*json)* format namely, students and faculty.

Student data:

Properties: first name, last name, gender, identity number, email address, phone number, IP address, data of birth, street address, language, city, street name and postal address.

Faculty data:

Properties: unique course identifier, course name, faculty name, unique subject identifier, subject description, subject cost.

Furthermore when the initial data was obtain, it was enriched with dimensions via a module within the prototype. Since the data model was already defined in the design phase of the prototype, the following properties were identified. Table. 5.6 provides a high level description of the different algorithms applied to populate the additional dimensions identified. After the algorithm was applied to the dimensions, the enriched data was stored in a pipe delimited files.

Additional data:

Properties: unique faculty identifier, subject term (in months), subject mark, subject graded symbol, gender title, street number, street address, postal code, province, unique country identifier, country name, unique student identifier, username, password, registration date, enrolment type.

Table 5.6: Enriched Data Generation Algorithm

Property	Algorithm
unique faculty identifier	Generated and concatenated three ASCII character codes from 65-90 which represents characters A-Z.
subject term (in months)	Generated a random number between 3 and 12.
subject mark	Generated a random number between 0 and 100.
subject graded symbol	Assigned a graded symbol based on the assigned subject mark : >= 80 : "A", >= 70 : "B", >= 60 : "C", >= 50 : "D", >= 40 : "E", >= 30 : "F", <= 29 : "G"
gender title	If the gender property is “Female” then assign “F” else assign “M”.
street number	Generated a random number between 0 and 150.
street address	Assigned postal address to street address
postal code	Generated a random number between 1000 and 4000.

province	Assigned a province base on the assigned country. See Appendix I.
country name	Assigned country based on the following list: "South Africa", "Angola", "Nigeria", "Namibia", "Botswana", "Egypt", "Tunisia"
unique country identifier	Based on randomly assigned country the following country codes apply: "South Africa" = "ZA", "Angola" = "AO", "Nigeria" = "NG", "Namibia" = "NA", "Botswana" = "BW", "Egypt" = "EG", "Tunisia" = "TN"
unique student identifier	Generated a random number between 100000000 and 99999999.
username	Concatenated the first letter of the student's last name and first name.
password	Generated an unique password based on the following character sets : "abcdefghijklmnopqursuvwxyz", "ABCDEFGHIJKLMNOPQRSTUVWXYZ" "123456789", "!@£$%^&*()#€""
registration date	Generated a date between the current date and 48 months in the past.
enrolment type	Generated a random number between 0 and 100. If the random number return an integer > 20 then assign "Full-Time" else assign "Part-Time"

5.4.3. Data Load

Once the data models were defined and the data prepared; loading the data into the native structures was the next step in the experimental set up. In essence, Extract, Transform, Load (ETL) approach was employed to facilitate this process of populating the underlying storage models. Before loading the enrich dataset into the native storage models, the physical schema was required. This was achieved programmatically via the prototype using the programming drivers and APIs to communicated with the locally install NoSQL database management systems.

The characteristics of each native storage models were taken into account, such as the particular language paradigms governing its creation and usage. The dataset is then loaded into memory from the pipe delimited files to an object structure representing the enrich data. For each of the native storage option, language-specific statement are created to insert the data set into the respective databases. Appropriate indexes were applied after the data load was completed for specific native databases to optimize query performance. This in itself had an impact on the experiment which is covered in section 5.5.

5.4.4. Data Metrics

This study is only concerned with the following metrics for data analysis:

Apdex:

The Apdex or Application Performance Index enabled the researcher to examine the satisfaction levels of participants in the experiment based on a score between 0 and 1. Zero being the worst possible score to achieve while 1 represent the best possible outcome. The number of participants directly correlates to the sample size of the Apdex metric. The satisfaction level for the unified query in the experiment was set at a target threshold of two seconds. The Apdex contained three categories to measure user satisfaction:

Table 5.7: Apdex Categories

#	Category	Description
1	Satisfied	The response time of the unified query is less than 2 seconds.
2	Tolerating	The response time of the unified query between 2 and (4 * 2) seconds.
3	Frustrating	The response time of the unified query is greater than (4 * 2) seconds.

According to the libraries documentation the formula to calculate the Apdex score is (app-metrics.io, 2021):

let's say

sr is satisfied requests

tr is tolerating requests

s is the total number of requests (i.e. sample size)

then

$$Apdex\ score = \left(sr + \left(\frac{tr}{2}\right)\right)/s$$

CPU Usage:

The CPU usage metric measured the time the CPU was busy processing the unified query producing a percentage value based on the total CPU available.

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 since the processor was initiated

then

$$CPU\ Usage = \frac{(et - st)}{(pa * pt)}$$

Memory Usage:

The memory usage metric was split into two categories as shown in Table. 5.9. These metrics measure the amount of memory consumed during the execution of the unified query platform.

Table 5.8: Memory Usage Categories

#	Category	Description
1	Virtual (VM)	The amount of memory accessed via the physical disk.
2	Physical (PM)	The amount of memory accessed via the machines RAM (Random Access Memory).

let's say

virtual memory:

ivm is the initial amount of virtual memory allocated to associate process

fvm is the final amount of virtual memory allocated to associate process

physical memory:

ipm is the initial amount of physical memory allocated to associate process

fpm is the final amount of physical memory allocated to associate process

then

$$virtual\ memory = fvm - ivm$$

$$physical\ memory = fpm - ipm$$

Query Execution Times:

The fundamental components of the unified query was subjected to performance tests, measure the amount of time taken to complete actions based on what has been described in Table. 5.9.

Table 5.9: Component Execution Categories

#	Category	Description
1	Parser	The time taken for the global parser to validate the query measured in milliseconds.
2	Translator	The time taken for the translator to generate the native NoSQL queries measured in milliseconds.
3	Executor	The time taken to execute the native query via the NoSQL drivers.

Error Rates:

Table. 5.10 describes how the error rate for each category are measured. The errors generated by the prototype for the unified query platform permits the researcher to know exactly where the failure occurred for a given unified query.

Table 5.10: Component Error Categories

#	Category	Description
1	Parser	The number of errors\exceptions the parser producers.
2	Translator	The number of errors\exceptions the translator producers.
3	Executor	The number of errors\exceptions the executor producers.

5.4.5. Automated Tests

The prototype incorporates a collection of autonomous tests to simulate human behaviour. The purpose of the automated tests was to assist the research in developing several scenarios to test the overall functionality of the unified query platform. Each automated test was assigned a

participant designated to perform a specific scenario. Setting up the scenarios delivered the researcher the required level of control in the test executions, providing a consistent and stable environment.

Collectively these tests were a key factor in evaluating if the unified query platform functions efficiently and in a reliable manner. Additionally, the automated tests encapsulated the results of each participant which aided the research to check its correctness. This procedures employed in this study enabled the researcher to conduct rigorous experiments, testing the system bounds of the unified query platform.

5.5 Experimental Results

According to Hevner et al 2004, the effectiveness of an artefact must be meticulously demonstrated by using the appropriated evaluation methods. As discussed in section 5.3 and experiment was carried out to access the utility of the unified query platform. The experiment executed ninety-one individual test cases whereby the metrics and query results were documented. Table. 5.11. shows a condensed view of the test cases executed in the experimented.

Table 5.11: Summary of Test Cases

#	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

5.5.1. Syntax and Sematic Validations

This group of participants performed basic syntax and sematic validations on the prototype's unified query (Appendix J). Participants were given the basic elements of unified query constructs and obliged to specify the query in a way that contradicts the rules governing the prototype's unified query. In certain cases, the unified query was not deemed as well-defined or the basic elements were not specified in order by the participants. While in other instances, fields were specific that was not defined in the unified queries metamodel.

Table 5.12: Syntax and Sematic Validations - Scenarios

Scenarios	Result
Incorrect syntax specified.	Syntax error occurred.
Incorrect fields specified.	Invalid fields specified: newproperty1, newproperty2
The basic query elements are not in the expected order.	Syntax error occurred.

Table 5.13: Syntax and Sematic Validations - Apdex

Test #	Frustrating	Tolerating	Satisfied	Score
87	0	0	1	1
88	0	0	1	1
89	0	0	1	1
90	0	0	1	1
91	0	0	1	1

Table 5.14: Syntax and Sematic Validations - CPU & Memory

Test #	CPU Usage (%)	VM(Bytes)	PM (Bytes)
87	0.000203429	65536	28672
88	0.000203428	65536	4096
89	0.000203428	0	0
90	0.000203428	0	0
91	0.000203428	0	0

Table 5.15: Syntax and Sematic Validations - Timers

Test #	Parser (ms)	Translator (ms)	Executor (ms)
87	6.755032094	-	-
88	6.78337294	-	-
89	6.802833852	-	-
90	6.844081612	-	-
91	7.018673661	-	-

Table 5.16: Syntax and Sematic Validations - Error Rate

Test #	Parser	Translator	Executor
87	-	-	1
88	-	-	1
89	-	-	1
90	-	-	1
91	1	-	-

5.5.2. Retrieve complete dataset

The goal was to interrogate the entire schema of the underlying native storage models and return the associated dataset. The participants attempted to perform a *Fetch* queries that retrieved all data from the all the supported NoSQL storage systems (Appendix K). The unified query specified no conditional statements except for a data restrictions in certain instances. The prototype generated native queries supported by the underlying storage driver which executed successfully, shown in Table 5.17. While the other tests results are not shown in that table below, the queries generated similar outputs.

Table 5.17: Retrieve complete dataset - Test Sample

Target	Generated Query
Redis	KEYS| *
Cassandra	SELECT title, idno, aka, initials, firstname, lastname, dob, genderid, address.streetno, address.streetname, address.postalcode, address.postalcode, address.city, address.country, email, cellno, studentno,

	registered.faculty, registered.course, registered.subject.descr, registered.subject.price, registered.subject.period, registered.registerdate, grades.subject, grades.marks, grades.symbol FROM student LIMIT 1000;
MongoDB	{ aggregate: 'students', pipeline: [{ $unwind : {path: '$enroll.subject'}},{ $project : { _id: '$_id', title : '$title', id_number : '$id_number', init : '$init', name : '$name', surname : '$surname', date_of_birth : '$date_of_birth', gender_identity : '$gender_identity', a_street : '$address.street', a_code : '$address.code', a_country : '$address.country', c_email_address : '$contact.email_address', c_phone : '$contact.phone', student_no : '$student_no', e_f_short_code : '$enroll.faculty.short_code', e_f_name : '$enroll.faculty.name', e_c_short_code : '$enroll.course.short_code', e_c_name : '$enroll.course.name', e_s_short_code : '$enroll.subject.short_code', e_s_name : '$enroll.subject.name', e_s_price : '$enroll.subject.price', e_s_duration : '$enroll.subject.duration', e_enrollment_type : '$enroll.enrollment_type', e_enrollment_date : '$enroll.enrollment_date'}},{$limit: 1000}],cursor: { }}
Neo4j	MATCH (pupi:pupil) WITH pupi MATCH (city:city)<-[:LIVES_IN]-(pupi)-[:CITIZEN_OF]->(coun:country) WITH pupi, city, coun MATCH (cour:course)<-[:ENROLLED_IN]-(pupi)-[:TRANSCRIPT]->(prog:progress) WITH pupi, city, coun, cour, prog MATCH (facu:faculty)<-[:OFFERED_IN]-(cour)-[:CONTAINS]->(subj:subject) WITH pupi, city, coun, cour, prog, facu, subj UNWIND apoc.convert.fromJsonList(prog.results) as res RETURN pupi.title, pupi.idnum, pupi.alias, pupi.initial, pupi.name, pupi.surname, pupi.dob, pupi.gender, city.description, coun.key, coun.description, pupi.email, pupi.mobile, pupi.studentnum, facu.key, facu.description, cour.key, cour.description, subj.key, subj.description, subj.cost, subj.term, res.subject, res.score, res.grade LIMIT 1000

Table 5.18: Retrieve complete dataset - Apdex

Test #	Frustrating	Tolerating	Satisfied	Score
1	0	1	0	1
9	0	1	0	1
28	0	1	0	1
45	0	0	1	1
66	0	1	0	1

Table 5.19: Retrieve complete dataset - CPU & Memory

Test #	CPU Usage (%)	VM(bytes)	PM (bytes)
1	0.000203434289649587	952733696	1123409920
9	0.000203428	915255296	1035886592
28	0.000203433	899678208	1028513792
45	0.000203432	333918208	543137792
66	0.00020343	318636032	537882624

Table 5.20: Retrieve complete dataset - Timers

Test #	Parser (ms)	Translator (ms)	Executor (ms)
1	0.699863213951067	0.700318929559155	0.700819814463631
9	0.755861983683434	0.755857418805569	0.756049445527891
28	0.903427925815745	0.903326589142185	0.903797082009779
45	1.1823484317691	1.18316836282101	1.1821881298606
66	1.29285453476938	5.17086469565111	5.17208321450023

Table 5.21: Retrieve complete dataset - Error Rate

Test #	Parser	Translator	Executor
1	-	-	-
9	-	-	-
28	-	-	-
45	-	-	-
66	-	-	-

5.5.3. Retrieve dataset where a single filter was applied

In the next set of tests, participants were instructed to employ a single condition to when querying data (Appendix L). Each native storage conceptually enforces these filters in a similar way. A number of variations were used with the aim of assessing how well the native query generators were able create executable queries. This relied heavily on the metamodel which binds to the schema of each native models influencing how the filters are applied to the dataset.

Table 5.22: Retrieve dataset where a single filter was applied - Test Sample

Target	Generated Query
Redis	GET|67101803610
Cassandra	SELECT title, idno, aka, initials, firstname, lastname, dob, genderid, address.streetno, address.streetname, address.postalcode, address.postalcode, address.city, address.country, email, cellno, studentno, registered.faculty, registered.course, registered.subject.descr, registered.subject.price, registered.subject.period, registered.registerdate FROM student WHERE idno = '67101803610' ALLOW FILTERING;
MongoDB	{ aggregate: 'students', pipeline: [{ $match : { id_number : '67101803610' }},{ $unwind : {path: '$enroll.subject'}},{ $project : { _id: '$_id', title : '$title', id_number : '$id_number', init : '$init', name : '$name', surname : '$surname', date_of_birth : '$date_of_birth', gender_identity : '$gender_identity', a_street : '$address.street', a_code : '$address.code', a_country : '$address.country', c_email_address : '$contact.email_address', c_phone : '$contact.phone', student_no : '$student_no', e_f_short_code : '$enroll.faculty.short_code', e_f_name : '$enroll.faculty.name', e_c_short_code : '$enroll.course.short_code', e_c_name : '$enroll.course.name', e_s_short_code : '$enroll.subject.short_code', e_s_name : '$enroll.subject.name', e_s_price : '$enroll.subject.price', e_s_duration : '$enroll.subject.duration', e_enrollment_type : '$enroll.enrollment_type', e_enrollment_date : '$enroll.enrollment_date'}}],cursor: { }}
Neo4j	MATCH (pupi:pupil) WITH pupi MATCH (city:city)<-[:LIVES_IN]-(pupi)-[:CITIZEN_OF]->(coun:country) WITH pupi, city, coun MATCH (cour:course)<-[:ENROLLED_IN]-(pupi) MATCH (facu:faculty)<-[:OFFERED_IN]-(cour)-[:CONTAINS]->(subj:subject) WITH pupi, city, coun, cour, facu, subj WHERE pupi.idnum = "67101803610" RETURN pupi.title, pupi.idnum, pupi.alias, pupi.initial, pupi.name, pupi.surname, pupi.dob, pupi.gender, city.description, coun.key, coun.description, pupi.email, pupi.mobile, pupi.studentnum, facu.key, facu.description, cour.key, cour.description, subj.key, subj.description, subj.cost, subj.term

Table 5.23: Retrieve dataset where a single filter was applied - Apdex

Test #	Frustrating	Tolerating	Satisfied	Score
2	0	0	1	1
3	0	1	0	1
4	0	0	1	1
10	0	0	1	1
16	0	1	0	1
17	0	0	1	1
54	0	0	1	1
67	0	0	1	1
77	0	0	1	1
78	0	0	1	1
79	0	0	1	1

Table 5.24: Retrieve dataset where a single filter was applied - CPU & Memory

Test #	CPU Usage (%)	VM(Bytes)	PM (Bytes)
2	0.000203433	919515136	1057464320
3	0.000203433	919449600	1056145408

4	0.000203432	915386368	1037086720
10	0.000203434	900005888	988917760
16	0.000203434	899940352	988184576
17	0.000203434	899940352	1028591616
54	0.000203431	319094784	538710016
67	0.00020343	317063168	741629952
77	0.000203429	578879488	762114048
78	0.000203429	577306624	728104960
79	0.000203429	577306624	757444608

Table 5.25: Retrieve dataset where a single filter was applied - Timers

Test #	Parser (ms)	Translator (ms)	Executor (ms)
2	0.725417023	0.725685535	0.725770987
3	0.729511974	0.729449715	0.729496134
4	0.750931779	0.750869246	0.750914423
10	0.818308132	0.818226444	0.818415541
16	0.848941559	0.848853868	0.848928108
17	0.88948269	0.889388597	0.889467468
54	1.241112071	1.240922368	1.241158599
67	1.365838095	5.46250671	5.463745806
77	1.629576833	6.517032522	6.518756728
78	1.70707669	6.826893089	6.82893176
79	1.803357054	7.211848996	7.214023897

Table 5.26: Retrieve dataset where a single filter was applied - Error Rate

Test #	Parser	Translator	Executor
2	-	-	-
3	-	-	-
4	-	-	-
10	-	-	-
16	-	-	-
17	-	-	-
54	-	-	-
67	-	-	-
77	-	-	1
78	-	-	1
79	-	-	1

5.5.4. Retrieve dataset where a multiples filters were applied

Participants in this part of the experiment extended test scenarios in section 5.5.2 by applying more than one filter to the to the query intent (Appendix M). The aimed was to reveal how the prototype was able to restrict the dataset when more than one condition were imposed on queries by participants.

Table 5.27: Retrieve dataset where a multiples filters were applied - Test Sample

Target	Generated Query
Redis	GET|67101803610
Cassandra	SELECT title, idno, aka, initials, firstname, lastname, dob, genderid, address.streetno, address.streetname, address.postalcode, address.postalcode, address.city, address.country, email, cellno, studentno, registered.faculty, registered.course, registered.subject.descr, registered.subject.price, registered.subject.period, registered.registerdate FROM student WHERE genderid IN ('F', AND idno IN ('67101803610');
MongoDB	{ aggregate: 'students', pipeline: [{ $match : { id_number : '67101803610', gender_identity : 'F' }},{ $unwind : {path: '$enroll.subject'}},{ $project : { _id: '$_id', title : '$title', id_number : '$id_number', init : '$init', name : '$name', surname : '$surname', date_of_birth : '$date_of_birth', gender_identity : '$gender_identity', a_street : '$address.street', a_code : '$address.code', a_country : '$address.country', c_email_address : '$contact.email_address',

	c_phone : '$contact.phone', student_no : '$student_no', e_f_short_code : '$enroll.faculty.short_code', e_f_name : '$enroll.faculty.name', e_c_short_code : '$enroll.course.short_code', e_c_name : '$enroll.course.name', e_s_short_code : '$enroll.subject.short_code', e_s_name : '$enroll.subject.name', e_s_price : '$enroll.subject.price', e_s_duration : '$enroll.subject.duration', e_enrollment_type : '$enroll.enrollment_type', e_enrollment_date : '$enroll.enrollment_date'}}],cursor: { }}
Neo4j	MATCH (pupi:pupil) WITH pupi MATCH (city:city)<-[:LIVES_IN]-(pupi)-[:CITIZEN_OF]->(coun:country) WITH pupi, city, coun MATCH (cour:course)<-[:ENROLLED_IN]-(pupi) MATCH (facu:faculty)<-[:OFFERED_IN]-(cour)-[:CONTAINS]->(subj:subject) WITH pupi, city, coun, cour, facu, subj WHERE pupi.idnum = "67101803610" AND pupi.gender = "F" RETURN pupi.title, pupi.idnum, pupi.alias, pupi.initial, pupi.name, pupi.surname, pupi.dob, pupi.gender, city.description, coun.key, coun.description, pupi.email, pupi.mobile, pupi.studentnum, facu.key, facu.description, cour.key, cour.description, subj.key, subj.description, subj.cost, subj.term

Table 5.28: Retrieve dataset where a multiples filters were applied - Apdex

Test #	Frustrating	Tolerating	Satisfied	Score
11	0	0	1	1
12	0	0	1	1
15	0	0	1	1
29	0	0	1	1
30	0	0	1	1
55	0	0	1	1
56	0	0	1	1
68	0	0	1	1
69	0	1	0	1
70	0	0	1	1
80	1	0	0	0
81	1	0	0	0

Table 5.29: Retrieve dataset where a multiples filters were applied - CPU & Memory

Test #	CPU Usage (%)	VM(Bytes)	PM (Bytes)
11	0.000203434	900005888	988688384
12	0.000203434	900005888	988655616
15	0.000203434	899940352	988184576
29	0.000203433	615460864	744546304
30	0.000203433	615329792	744284160
55	0.000203431	319094784	538652672
56	0.000203431	319029248	538607616
68	0.00020343	310771712	741445632
69	0.00020343	310771712	741896192
70	0.00020343	579207168	762544128
80	0.000203429	575733760	758886400
81	0.000203429	307298304	494813184

Table 5.30: Retrieve dataset where a multiples filters were applied - Timers

Test #	Parser (ms)	Translator (ms)	Executor (ms)
11	0.823237238	0.823164432	0.823238242
12	0.82833822	0.828254384	0.828317399
15	0.843936171	0.843850231	0.843918157
29	0.997038414	0.996928902	0.997432128
30	1.004696105	1.004579686	1.00473217
55	1.243998395	1.243804839	1.243922862
56	1.246823517	1.246629105	1.246742022
68	1.384250142	5.53613422	5.537546084
69	1.403454922	5.61293171	5.614385662
70	1.537482434	6.148871197	6.150464124
80	1.908067914	7.630522787	7.632712962

81	2.852472965	11.40603636	11.41123184

Table 5.31: Retrieve dataset where a multiples filters were applied - Error Rate

Test #	Parser	Translator	Executor
11	-	-	1
12	-	-	1
15	-	-	1
29	-	-	-
30	-	-	-
55	-	-	-
56	-	-	-
68	-	-	1
69	-	-	1
70	-	-	1
80	-	-	1
81	-	-	1

5.5.5. Apply a limit to the dataset retrieval process

Restrictions were stated in the unified query platform; which is support by each targeted storage model except Redis. Participants specified the data limits ensuring the data returned reflect the amount of records requested to be returned (Appendix N). The purpose of these tests was to enable participants to control the number of data being returned.

Table 5.32: Apply a limit to the dataset retrieval process - Test Sample

Target	Generated Query
Redis	-
Cassandra	SELECT title, idno, aka, initials, firstname, lastname, dob, genderid, address.streetno, address.streetname, address.postalcode, address.postalcode, address.city, address.country, email, cellno, studentno, registered.faculty, registered.course, registered.subject.descr, registered.subject.price, registered.subject.period, registered.registerdate, grades.subject, grades.marks, grades.symbol FROM student LIMIT 10;
MongoDB	{ aggregate: 'students', pipeline: [{ $unwind : {path: '$enroll.subject'}},{ $project : { _id: '$_id', title : '$title', id_number : '$id_number', init : '$init', name : '$name', surname : '$surname', date_of_birth : '$date_of_birth', gender_identity : '$gender_identity', a_street : '$address.street', a_code : '$address.code', a_country : '$address.country', c_email_address : '$contact.email_address', c_phone : '$contact.phone', student_no : '$student_no', e_f_short_code : '$enroll.faculty.short_code', e_f_name : '$enroll.faculty.name', e_c_short_code : '$enroll.course.short_code', e_c_name : '$enroll.course.name', e_s_short_code : '$enroll.subject.short_code', e_s_name : '$enroll.subject.name', e_s_price : '$enroll.subject.price', e_s_duration : '$enroll.subject.duration', e_enrollment_type : '$enroll.enrollment_type', e_enrollment_date : '$enroll.enrollment_date'}},{$limit: 10}],cursor: { }}
Neo4j	MATCH (pupi:pupil) WITH pupi RETURN pupi.title, pupi.idnum, pupi.alias, pupi.initial, pupi.name, pupi.surname, pupi.dob, pupi.gender, pupi.email, pupi.mobile LIMIT 100

Table 5.33: Apply a limit to the dataset retrieval process - Apdex

Test #	Frustrating	Tolerating	Satisfied	Score
13	0	0	1	1
31	0	0	1	1
46	0	0	1	1
47	0	0	1	1
48	0	0	1	1
49	0	0	1	1
50	0	0	1	1
51	0	0	1	1
52	0	0	1	1

53	0	0	1	1

Table 5.34: Apply a limit to the dataset retrieval process - CPU & Memory

Test #	CPU Usage (%)	VM(Bytes)	PM (Bytes)
13	0.000203434	988651520	900005888
31	0.000203433	745390080	615329792
46	0.000203432	538890240	320733184
47	0.000203431	538886144	320733184
48	0.000203431	538824704	319160320
49	0.000203431	538779648	319160320
50	0.000203431	538779648	319160320
51	0.000203431	538779648	319160320
52	0.000203431	538779648	319160320
53	0.000203431	538759168	319094784

Table 5.35: Apply a limit to the dataset retrieval process - Timers

Test #	Parser (ms)	Translator (ms)	Executor (ms)
13	0.833045067	0.832973656	0.833040369
31	1.011702719	1.011577607	1.011679184
46	1.199600382	1.199428542	1.19956104
47	1.208681877	1.208504997	1.208635624
48	1.21618074	1.216000246	1.216131979
49	1.223323951	1.223139826	1.223272824
50	1.229190657	1.229004693	1.229139689
51	1.232823066	1.232639427	1.232766074
52	1.235600599	1.235406371	1.23552967
53	1.238518221	1.238326487	1.238451466

Table 5.36: Apply a limit to the dataset retrieval process - Error Rate

Test #	Parser	Translator	Executor
13	-	-	-
31	-	-	-
46	-	-	-
47	-	-	-
48	-	-	-
49	-	-	-
50	-	-	-
51	-	-	-
52	-	-	-
53	-	-	-

5.5.6. Apply sorting to the dataset retrieval process

In this simulation group, the participants specified that one or more fields on the dataset to be sorted (Appendix O). The query parser component enabled participant, not only to specify the sort field but also the sorting direction.

Table 5.37: Apply sorting to the dataset retrieval process - Test Sample

Target	Generated Query
Redis	GET|67101803610
Cassandra	SELECT title, idno, aka, initials, firstname, lastname, dob, genderid, address.streetno, address.streetname, address.postalcode, address.postalcode, address.city, address.country, email, cellno, studentno, registered.faculty, registered.course, registered.subject.descr, registered.subject.price, registered.subject.period, registered.registerdate FROM student WHERE genderid IN ('M') AND idno IN ('67101803610', AND studentno IN ('979883209') ORDER BY lastname ASC , idno ASC ;
MongoDB	{ aggregate: 'students', pipeline: [{ $match : { $or : [{ id_number : '67101803610' }, { gender_identity : 'M' }], student_no : '979883209' }},{

	$unwind : {path: '$enroll.subject'}},{ $project : { _id: '$_id', title : '$title', id_number : '$id_number', init : '$init', name : '$name', surname : '$surname', date_of_birth : '$date_of_birth', gender_identity : '$gender_identity', a_street : '$address.street', a_code : '$address.code', a_country : '$address.country', c_email_address : '$contact.email_address', c_phone : '$contact.phone', student_no : '$student_no', e_f_short_code : '$enroll.faculty.short_code', e_f_name : '$enroll.faculty.name', e_c_short_code : '$enroll.course.short_code', e_c_name : '$enroll.course.name', e_s_short_code : '$enroll.subject.short_code', e_s_name : '$enroll.subject.name', e_s_price : '$enroll.subject.price', e_s_duration : '$enroll.subject.duration', e_enrollment_type : '$enroll.enrollment_type', e_enrollment_date : '$enroll.enrollment_date'}},{$sort: {surname : 1 , id_number : 1 }}],cursor: { }}
Neo4j	MATCH (pupi:pupil) WITH pupi MATCH (city:city)<-[:LIVES_IN]-(pupi)-[:CITIZEN_OF]->(coun:country) WITH pupi, city, coun MATCH (cour:course)<-[:ENROLLED_IN]-(pupi) MATCH (facu:faculty)<-[:OFFERED_IN]-(cour)-[:CONTAINS]->(subj:subject) WITH pupi, city, coun, cour, facu, subj WHERE pupi.idnum = "67101803610" OR pupi.gender = "M" AND pupi.studentnum = "979883209" RETURN pupi.title, pupi.idnum, pupi.alias, pupi.initial, pupi.name, pupi.surname, pupi.dob, pupi.gender, city.description, coun.key, coun.description, pupi.email, pupi.mobile, pupi.studentnum, facu.key, facu.description, cour.key, cour.description, subj.key, subj.description, subj.cost, subj.term ORDER BY pupi.surname ASC, pupi.idnum ASC

Table 5.38: Apply sorting to the dataset retrieval process - Apdex

Test #	Frustrating	Tolerating	Satisfied	Score
14	0	0	1	1
32	0	1	0	1
33	0	0	1	1
34	0	0	1	1
35	0	0	1	1
36	0	0	1	1
57	0	0	1	1
71	0	0	1	1

Table 5.39: Apply sorting to the dataset retrieval process - CPU & Memory

Test #	CPU Usage (%)	VM(Bytes)	PM (Bytes)
14	0.000203434	900005888	988622848
32	0.000203433	615329792	745381888
33	0.000203432	332476416	543109120
34	0.000203432	332476416	543096832
35	0.000203432	332476416	544456704
36	0.000203432	334049280	544477184
57	0.000203431	318767104	539201536
71	0.00020343	579207168	762544128

Table 5.40: Apply sorting to the dataset retrieval process - Timers

Test #	Parser (ms)	Translator (ms)	Executor (ms)
14	0.838279989	0.838233091	0.838366485
32	1.019247682	1.019120228	1.019263099
33	1.135520321	1.135370823	1.135494587
34	1.145365865	1.145243427	1.145369575
35	1.155385082	1.155229619	1.155394427
36	1.166003315	1.165874238	1.166003396
57	1.250216694	1.250023287	1.250215787
71	1.563632595	6.253444736	6.255122

Table 5.41: Apply sorting to the dataset retrieval process - Error Rate

Test #	Parser	Translator	Executor

14	-	-	1
32	-	-	-
33	-	-	-
34	-	-	-
35	-	-	-
36	-	-	-
57	-	-	-
71	-	-	1

5.5.7. Aggregation on a datasets

Participants commanded the unified queries to perform various aggregated functions on the existing datasets (Appendix P). The aggregation could be only specified as part of the *Fetch* clause properties on a set of values. The intent is to condense a set of values into a single calculated value. Participants only used these functions in the following circumstances:

- NCOUNT - when counting the number of rows on a field.
- NSUM - adding the numerical values of a specified field together.
- NMIN - retrieving the lowest value in the set of values on a particular field.
- NMAX - retrieving the highest value in the set of values on a particular field.
- NAVG - calculating the average values based in set of values returned.

Table 5.42: Aggregation on a datasets - Test Sample

Target	Generated Query
Redis	GET|21708702176
Cassandra	SELECT idno, initials, firstname, lastname, SUM(grades.marks) as g_marks FROM student WHERE idno = '21708702176' ALLOW FILTERING;
MongoDB	{ aggregate: 'students', pipeline: [{ $match : { id_number : '58602700606' }},{ $unwind : {path: '$enroll.subject'}},{ $project : { _id: '$_id', id_number : '$id_number', init : '$init', name : '$name', surname : '$surname', subject : '$enroll.subject'}},{ $group : { _id: '$_id', id_number : { '$first' : '$id_number'}, init : { '$first' : '$init'}, name : { '$first' : '$name'}, surname : { '$first' : '$surname'}, s_price: { $sum: '$subject.price'}}}],cursor: { }}
Neo4j	MATCH (pupi:pupil) WITH pupi MATCH (prog:progress)<-[:TRANSCRIPT]-(pupi) UNWIND apoc.convert.fromJsonList(prog.results) as res WITH pupi, res WHERE pupi.idnum = "21708702176" RETURN pupi.idnum, pupi.initial, pupi.name, pupi.surname, SUM(res.score) as score

Table 5.43: Aggregation on a datasets - Apdex

Test #	Frustrating	Tolerating	Satisfied	Score
18	0	0	1	1
19	0	0	1	1
20	0	0	1	1
21	0	0	1	1
22	0	0	1	1
37	0	0	1	1
38	0	0	1	1
39	0	0	1	1
40	0	0	1	1
41	0	0	1	1
58	0	0	1	1
59	0	0	1	1
60	0	0	1	1
61	0	0	1	1
62	0	0	1	1
72	0	0	1	1
73	0	0	1	1

74	0	0	1	1
75	0	0	1	1
76	0	0	1	1

Table 5.44: Aggregation on a datasets - CPU & Memory

Test #	CPU Usage (%)	VM(Bytes)	PM (Bytes)
18	0.000203434	899940352	1029087232
19	0.000203434	899874816	1028747264
20	0.000203433	899874816	1028747264
21	0.000203433	899809280	1028706304
22	0.000203433	899743744	1028657152
37	0.000203432	335622144	544497664
38	0.000203432	335622144	543531008
39	0.000203432	335622144	543490048
40	0.000203432	335556608	543469568
41	0.000203432	335556608	543444992
58	0.000203431	318767104	539164672
59	0.000203431	318767104	538882048
60	0.00020343	318767104	538247168
61	0.00020343	318767104	538206208
62	0.00020343	318767104	538157056
72	0.00020343	579207168	762507264
73	0.00020343	579207168	762433536
74	0.000203429	579207168	762380288
75	0.000203429	579207168	762363904
76	0.000203429	579207168	762241024

Table 5.45: Aggregation on a datasets - Timers

Test #	Parser (ms)	Translator (ms)	Executor (ms)
18	0.893656921	0.89359346	0.893795558
19	0.894454023	0.894366719	0.894433827
20	0.894983046	0.894876652	0.894943118
21	0.895515273	0.895409303	0.895476333
22	0.89604188	0.895936584	0.896002538
37	1.16716482	1.166990141	1.167314267
38	1.168287499	1.168111197	1.168260032
39	1.169303526	1.169127095	1.169273718
40	1.170235607	1.170059616	1.170168914
41	1.171168191	1.170995799	1.171110667
58	1.252296723	1.25209541	1.252275625
59	1.257363089	1.257163027	1.257299425
60	1.263207194	1.263006052	1.263112447
61	1.268023555	1.267819842	1.267934575
62	1.273273995	1.273067771	1.273174284
72	1.578624909	6.313261195	6.314792484
73	1.588028804	6.350857337	6.352112004
74	1.597840904	6.390106377	6.391008124
75	1.607189223	6.427486486	6.42915054
76	1.618573017	6.473022826	6.474262152

Table 5.46: Aggregation on a datasets - Error Rate

Test #	Parser	Translator	Executor
18	-	-	-
19	-	-	-
20	-	-	-
21	-	-	-
22	-	-	-
37	-	-	-
38	-	-	-
39	-	-	-
40	-	-	-

41	-	-	-
58	-	-	-
59	-	-	-
60	-	-	-
61	-	-	-
62	-	-	-
72	-	-	-
73	-	-	-
74	-	-	-
75	-	-	-
76	-	-	-

5.5.8. Update existing dataset

The test sample below, Table 5.47, shows how the differentiating NoSQL code generators produced modification statements for underlying storage models. Participants performed these tests on a subset of data within each NoSQL store guided by the base data used in the data load process, section 5.4.3 (Appendix Q). The purpose of this test was to update on one or more fields. Furthermore, the conditional statements set in the unified query played in important role in determining whether are not the generated native queries would be accepted or not.

Table 5.47: Update existing dataset - Test Sample

Target	Generated Query
Redis	GET|34502402028||;SET|34502402028||%gender=M
Cassandra	MODIFY { student } PROPERTIES { name = 'Test 1', surname = 'Test 2', initial = 'TT'} FILTER_ON { identifier = '5' } TARGET { cassandra }
MongoDB	{ update: 'students',updates: [{q:{ id_number : '83604407222' }, u: {$set: {name : 'Jane', surname : 'Doe', init : 'JD'}}}]}
Neo4j	MATCH (pupi:pupil) WITH pupi WHERE pupi.idnum = "83604407222" SET pupi.name= "Jane", pupi.surname= "Doe", pupi.initial= "JD"

Table 5.48: Update existing dataset - Apdex

Test #	Frustrating	Tolerating	Satisfied	Score
5	0	0	1	1
6	0	0	1	1
23	0	0	1	1
24	0	0	1	1
25	0	0	1	1
42	0	0	1	1
43	0	0	1	1
63	0	0	1	1
64	0	0	1	1
82	0	0	1	1
83	0	0	1	1
84	0	0	1	1
85	0	0	1	1

Table 5.49: Update existing dataset - CPU & Memory

Test #	CPU Usage (%)	VM(Bytes)	PM (Bytes)
5	0.000203431	915386368	1037041664
6	0.000203431	915320832	1036414976
23	0.000203433	899743744	1028595712
24	0.000203433	899743744	1028579328
25	0.000203433	899743744	1028575232
42	0.000203432	335491072	543395840
43	0.000203432	335491072	543285248
63	0.00020343	318701568	538128384

64	0.00020343	318636032	538058752
82	0.000203429	65536	37429248
83	0.000203429	65536	37421056
84	0.000203429	65536	36290560
85	0.000203429	65536	32768

Table 5.50: Update existing dataset - Timers

Test #	Parser (ms)	Translator (ms)	Executor (ms)
5	0.751188216	0.751151683	0.751209113
6	0.751793928	0.751722453	0.751760518
23	0.896530068	0.896423185	0.896520147
24	0.896834127	0.896727531	0.896826733
25	0.89728965	0.897181775	0.897271576
42	1.171949394	1.171773513	1.171921426
43	1.17268572	1.172510024	1.172613813
63	1.278380189	1.278172	1.278383654
64	1.279771416	1.279565	1.279667931
82	5.699773656	22.78313456	22.79966534
83	5.758787651	23.01965019	23.0454284
84	5.905735194	23.60591669	23.63827692
85	-	-	-

Table 5.51: Update existing dataset - Error Rate

Test #	Parser	Translator	Executor
5	-	-	-
6	-	-	-
23	-	-	-
24	-	-	1
25	-	-	1
42	-	-	-
43	-	-	-
63	-	-	-
64	-	-	-
82	-	-	1
83	-	-	1
84	-	-	-
85	1	-	-

5.5.9. Data inserts

Participants were provided conditional instructions to added new data to the existing storage models (Appendix R). These instructions were:

- Insert data referencing one or more non-indexed fields which included the primary key.
- Insert data referencing one or more indexed fields with the primary key
- Insert data referencing one or more fields (indexed and non-indexed) without the primary key.

Table 5.52: Data inserts - Test Sample

Target	Generated Query
Redis	GET|34502402028||;SET|34502402028||%gender=M
Cassandra	INSERT INTO student(id, idno, title, firstname, lastname, studentno) VALUES ('323323995', '876765564431', 'Miss', 'Lauren', 'Cole', '7149222') ;
MongoDB	{ insert: 'students',documents: [{id_number : '6062390', title : 'Miss', name : 'Lauren', surname : 'Cole', student_no : '53012'}]}
Neo4j	CREATE (pupi:pupil { idnum : "8078891", title : "Miss", name : "Lauren", surname : "Cole" })

Table 5.53: Data inserts - Apdex

Test #	Frustrating	Tolerating	Satisfied	Score
7	0	0	1	1
8	0	0	1	1
26	0	0	1	1
27	0	0	1	1
44	0	0	1	1
65	0	0	1	1
86	0	0	1	1

Table 5.54: Data inserts - CPU & Memory

Test #	CPU Usage (%)	VM(Bytes)	PM (Bytes)
7	0.00020343	915255296	1036345344
8	0.000203429	915255296	1036210176
26	0.000203433	899743744	1028571136
27	0.000203433	899743744	1028530176
44	0.000203432	335491072	543285248
65	0.00020343	318636032	538034176
86	0.000203429	65536	28672

Table 5.55: Data inserts - Timers

Test #	Parser (ms)	Translator (ms)	Executor (ms)
7	0.752062291	0.752000786	0.752043709
8	0.752182168	0.752111545	-
26	0.897776388	0.897670266	0.897763045
27	0.898043596	0.897937356	0.898008268
44	1.173134963	1.172958539	1.173086296
65	1.28123429	1.281026048	1.281167449
86	-	-	-

Table 5.56: Data inserts - Error Rate

Test #	Parser	Translator	Executor
7	-	-	-
8	-	-	1
26	-	-	-
27	-	-	1
44	-	-	-
65	-	-	-
86	1	-	-

5.6 Summary

This chapter provides the reader a detailed description of the experimental activities undertaken to assess the efficiency and effectiveness of the unified query platform. The experimental endeavour was informed by guideline 5 (section 3.4.5) of the DSR strategy. The chapter starts by providing the reader an overview of the experiment, identifying the participants and the procedures utilised to evaluate the prototype within the controlled test environment.

The researcher discussed how the participants executed varying test scenarios subjected to different conditions. Fundamentally, the test scenarios covered the three commands used throughout the experiments, namely: *Fetch*, *Modify* and *Add.* The automated tests ranged in the instance of *Fetch* commands ranged from simple to more complex queries that required more system resources. The purpose of the experiment articulated and the results documented aimed

to provide key insights on the inner workings of the prototype which is covered in the next chapter.

CHAPTER SIX : FINDINGS AND DISCUSSIONS

6.1 Introduction

In this chapter the researcher presents the results of the experiment conducted on the prototype as stated in chapter 5. This study commenced by identifying the research problem for unified query platforms. Subsequently, the researcher set the aim of the endeavour defining actionable objectives which essentially served as the motivation for the prototype development. The research questions posed in the study was the driving force to seek answers when developing unified query platforms using a polyglot persistent approach. Moreover, this chapter aims to provide a clear and concise summation of the what was discovered as well as the implications and significance of the research findings.

6.2 Research Questions

The research outputs produced by the prototype and the systematic literature review conducted in chapter 2 answers the primary research questions inquired in section 1.5. as shown in Table 6.1. In chapter 2, the reader's focus was drawn to research questions RQ1 and RQ2, to provide context and the necessary design and architectural principles when developing a unified query platform. The chapter starts with the theoretical aspects to the essential components required to develop such a system. Chapter 4 proceeds to demonstrate how sub question, RQ3, was addressed. Essentially building the prototype based on the fundamental learnings gathered in systematic literature review while making inferences to realise the solution (Vaishnavi, Kuechler & Petter, 2019). The entire process was guided by the DSR process models adhering to the guidelines discussed in Chapter 3 – specifically guidelines 3 and 4 (section 3.4.3 and section 3.4.4).

Table 6.1: Primary Research Questions

#	Research Question
1	How can a unified query platform be developed for the four primary categories of NoSQL databases using polyglot persistent technique?

Seeking answers to the these questions informed the primary research question as each of the RQ's was align to specific objectives. Chapter 5, motivated by RQ 4, proceeded to evaluated the utility of the prototype construct addressing how well the prototype implemented the foundations, design and architectural principles of the unified query platform. The objectives associated with the sub research questions was to determine a set of guidelines; making informed design and architectural choices when creating an unified query platform. Data collected from the literature review process interrogated published articles, journals and conference papers where these types of systems were constructed using a polyglot approach commonly known as middleware. Several papers discussed the varying methods applied in building a unified query platform (Koutroumanis et al., 2021; Zhang et al., 2021; Ramadhan et al., 2020). A common thread was determined whereby this study summarized the findings. While authors of these papers may have different terminology in certain instances, the respective fundamental concepts are the same.

Once the essential guidelines for a unified query platform was determined, the researcher address the second research question to determine which design and architectural principles should apply in the construction process of the prototype. Before embarking on the actually system inner workings of a unified query platform, careful planning was carried out to evaluate the data landscape which the creator wanted to abstract (Kolonko & Müllenbach, 2020). De facto design and architectural principles for unified queries directed the development process (see section 2.6). Therefore it was important to determine which widely accepted industry practices should apply to these systems.

RQ3 addressed the degree at which an abstract query can be translated into native queries. While the research outputs discovered during the systematic literature informed the construction phase of the prototype, insights were attained during the simulated tests conducted on the prototype by the automated participants. The influencing factors affecting the capabilities of the prototype invariably depended on the complexity of the unified query; the underlying native storage models the supported features for each of the target databases (Cox et al. 2020; Ramadhan et al., 2020).

6.3 Results of the Prototype Evaluation

The next section discusses the results obtained from the automates test simulations. The researcher imposed a number of varying tests on the prototype to determine the implementations efficiency and effectiveness. The participant groups (PG) were tasked to execute pre-determined queries encapsulating a full spectrum of scenarios. Participants were able to choose which storage models to target. This was determine how well each aspect of the unified query performed individually as well as a collective (see Appendix J through to R).

6.3.1. Syntax and Sematic Validations

To establish the initial boundaries of unified query, participants in this test group were intentionally assign queries that were expected fail. These queries were either not syntactically correct or the order of the query elements where specified incorrectly. In some cases the participants specified data fields that was not supported by the unified data model. In each of the cases, except for test case 89, the prototype correctly produced an error stating that "*Syntax error occurred*". However, even though the context of the error message given to the user, it does not hone in on the exact cause of the error. Test case 89 provides a more perceptive message stating exactly what was wrong with the query by stating "*Invalid fields specified: newproperty1,newproperty2*".

Figure. 6.1: PG 1 - Errors

Based on the metrics produced, the queries adheres to consuming the minimum possible amount resources in the call stack which resulted in an optimal execution path. This is also reflected in the Apdex scores as each one indicated that the result produced was given in the acceptable timeframe.

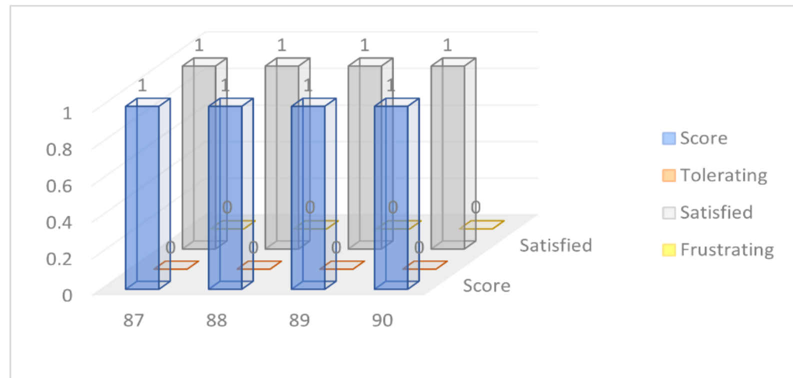
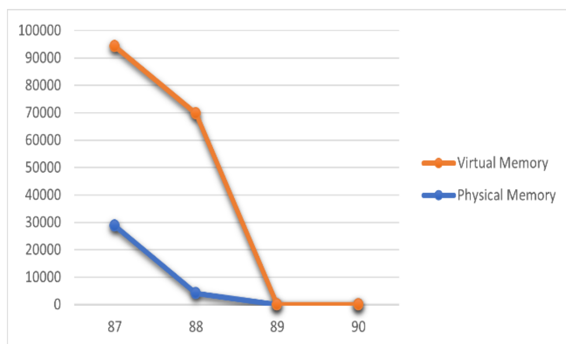


Figure. 6.2: PG 1 - Apdex Scores



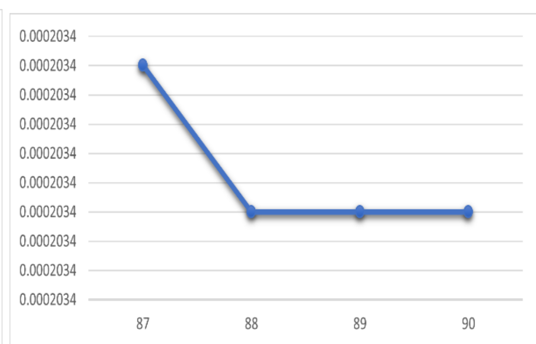


Figure. 6.3: PG 1 - Memory Allocations

Figure. 6.4: PG 1 - CPU Processing Time

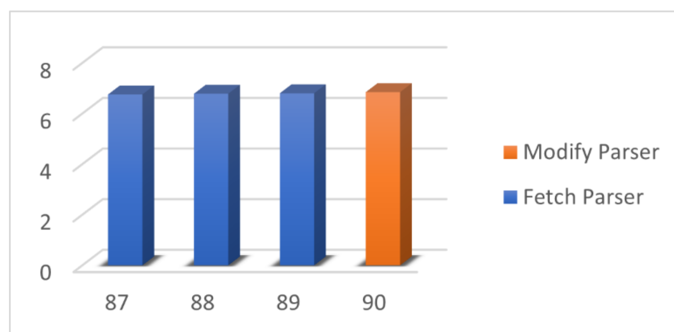


Figure. 6.5: PG 1 - Parser Failures

These tests were expected to fail upfront as shown in Figure 6.5, hence the prototype did not perform any translations or executions. The research observed that the error was documented by the execution metric instead of the parser metric (except test case 89). This is due to the fact that error threw an exception which was handled by the parser and delegated back to the calling

method hiding the actual parser error. This was not the expected behaviour from a logging standpoint as this error should have been logged by the parser.

6.3.2. Retrieve complete dataset

The goal for this test group was to determine if all the full results was returned when a participant request the all fields supported in the unified data model. As shown in Appendix E, not all native fields were mapped to the unified model since some of the properties were not present in the native storage schemas. Thus, the researcher expected that certain fields would not be mapped to the value of the unified data model. The prototype enabled the supported storage options to be specified interchangeably, providing participants the flexibility to target native databases they wanted to interrogate.

The participants started off by targeting only one of the supported NoSQL databases to determine how well the results returned are mapped for each storage option. The researcher observed all the native fields in Redis and Cassadra, were able to successful bind to unified results model. In the case of MongoDB, the “postalcode” field was not able to bind and was unable to map the country “name” field in the case of the Neo4j. While the results in both scenarios were successfully retrieved and available, a bug in the code prevented these values to be presented to the participants. The particular bug was not localized to these tests only but rather to all simulations where these fields were specified in the “*Fetch*” intent. The prototype failed to detect and report this issue in the reporting metrics.

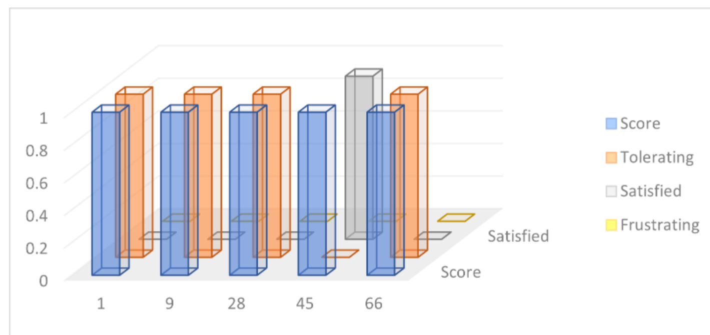
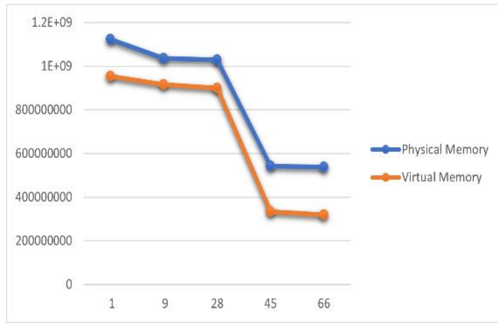


Figure. 6.6: PG 2 - Apdex Scores

The Apdex indicated that the execution path of the unified query was deemed acceptable and provided the result in an acceptable timeframe (Figure 6.6). It should be noted that most for the test scenarios were classified as tolerable. Test case 45, which targeted the Neo4j database, was the only scenario whereby the reporting metrics classified the unified query to be satisfied. However this was due a row restriction placed on the query as preliminary checks discovered that retrieving a large amount of connected nodes degraded performance drastically as observed by Cox et al. (2020).



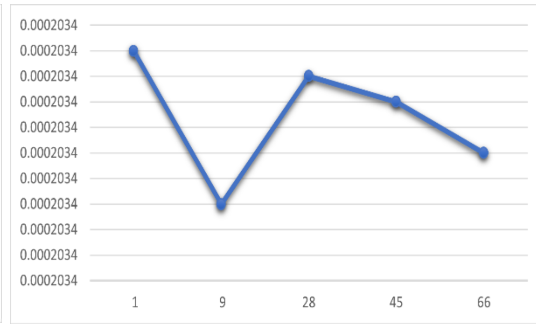


Figure. 6.7: PG 2 - Memory Allocations **Figure. 6.8**: PG 2 - CPU Processing Time

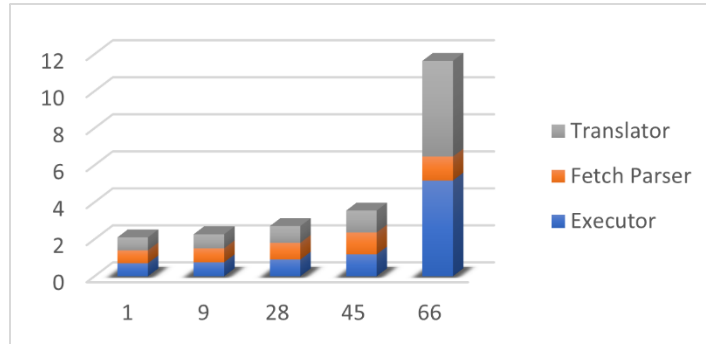


Figure. 6.9: PG 2 - Parser, Translator and Executor Times

The test scenarios produced no errors as expected by the researcher other than the mapping errors previously mentioned. The researcher observed that while Redis target generally outperformed its counterparts in terms of speed, it was allocated more memory and held onto the CPU longer since that storage model is not able to compress data as well as the other target stores, i.e. Redis has more individual records (Figure 6.7 and 6.8). Over a Gigabyte was allocated to the process indicating perhaps a need to hash the values to reduce the its footprint.

Neo4j, although within the acceptable Apdex range, it took much longer when compared to the other target stores to produce a result. Due to its intrinsic complex nature, connecting entities or nodes to one other via vertices and edges, the prototype required a more complex algorithm to translate the unified query to the corresponding native query, as shown in Figure 6.9. Even though, it took longer to produce a result, the CPU and memory footprint within the prototype was relatively low as most of the work was delegated to the database. The Cassandra and MongoDB targets did not show a significant variation in performance to the forementioned stores and produced results reliably and efficiently.

6.3.3. Retrieve dataset where a single filter was applied

Filtering plays an important role in large datasets. Participants added a single filter to the "*Fetch*" intent to determined how well each of the targeted storage translators are able to successfully generate the respective native queries and how well the data matched the intent. Once the filters were applied, the results from simulations highlighted quite a few inconsistencies, Figure 6.10.

The general efficiency of the prototype demonstrated that the overall performance was acceptable, Figure 6.11. However, a number of use cases produced the incorrect outputs.

Figure. 6.10: PG 3 - Errors

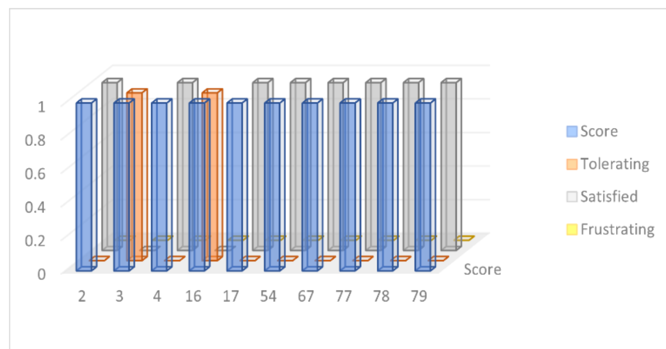
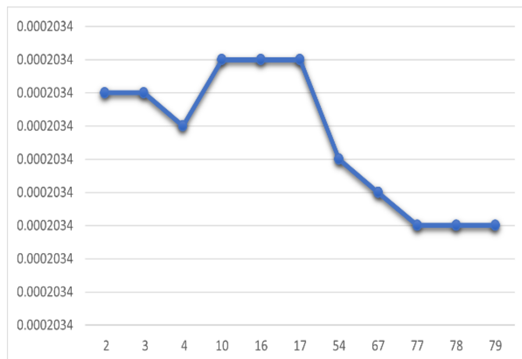


Figure. 6.11: PG 3 - Apdex Scores

Test case 3 demonstrated that if the filter is not applied on the Redis key, the full result set returned. Figure 6.11 and 6.12 respective shows that Redis consumed more resources as a result of these cases. Test cases 77, 78 and 79 produced partially correct results. In all three instances the native query's generator for Cassandra was not able to executed successfully. This is due to system constraints imposed on the Cassandra database management system. The reason the generated query produced errors was due to the prototype not following the search index principles Cassandra enforces. The researcher assert's that the prototype should prevent a query from being generated if the search indexes are not present in the unified query. Based on the empirical data, this participant group demonstrated that the feature mismatches either results in the misuse of the machines resources or invalid output are generated.



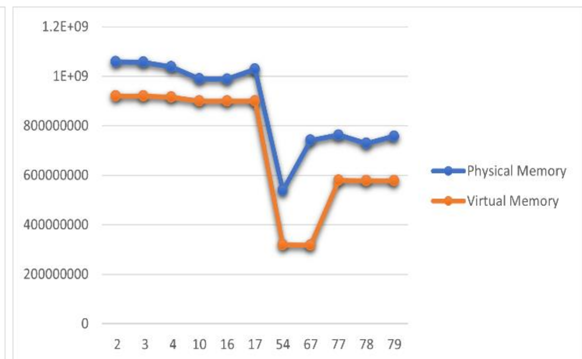


Figure. 6.12: PG 3 - CPU Processing Time

Figure. 6.13: PG 3 - Memory Allocations

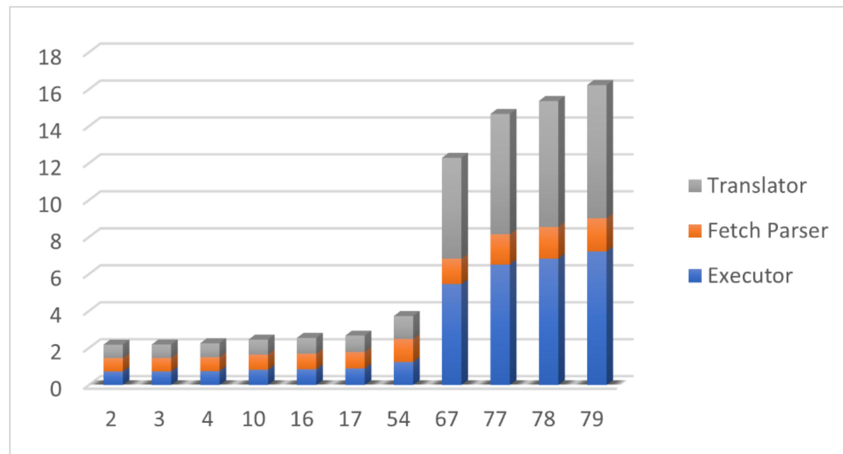


Figure. 6.14: PG 3 - Parser, Translator and Executor Times

6.3.4. Retrieve dataset where a multiples filters were applied

The simulated tests conducted by participants in this group compounded the errors and inaccuracies observed in section 6.3.3, resulting in more errors being generated by the prototype shown 6.15. In these scenarios the "*AND\OR*" operators were utilised to control the result set to very specific data points. The failed tests were localised to specific native storage systems as the prototype created native queries that was not assimilated to rules governing the these storage option.

Figure. 6.15: PG 4 - Multiple Filter Errors

The first notable observation relates to how individual native queries; once an "OR" operator was applied to the search filter, the Apdex score was negatively affected; depending on the ontology and the relations within the data resulted in performance degradation. This is highlighted in Figure 6.16 where test case 69, 80 and 81 experienced a performance degradation.

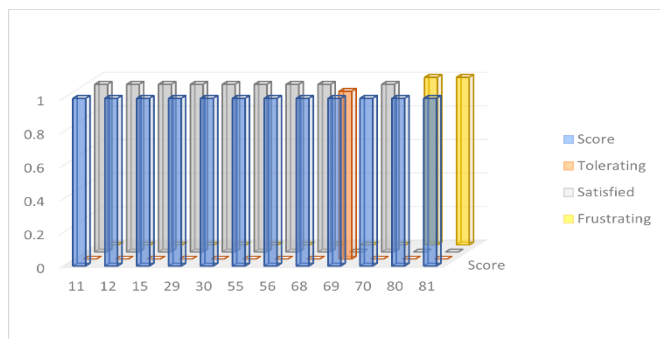


Figure. 6.16: PG 4 - Apdex Scores

Based on the error rate, the prototype generated the native queries which consistently failed to executed on the specific storage mechanisms. For the Redis executable queries, the prototype determined if one of the filters was configured as key index. If so, it applied the filter on the recognised index and ignored any other filters.

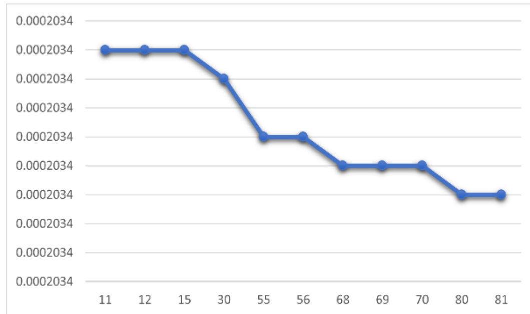


Figure. 6.17: PG 4 - CPU Processing Time

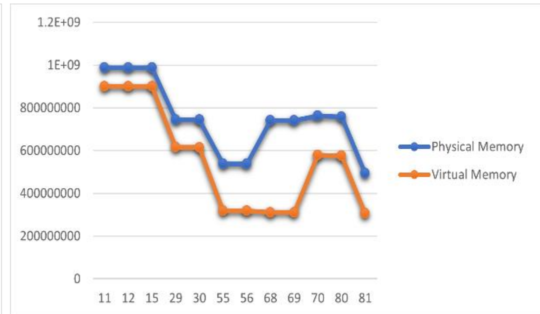


Figure. 6.18: PG 4 - Memory Allocations

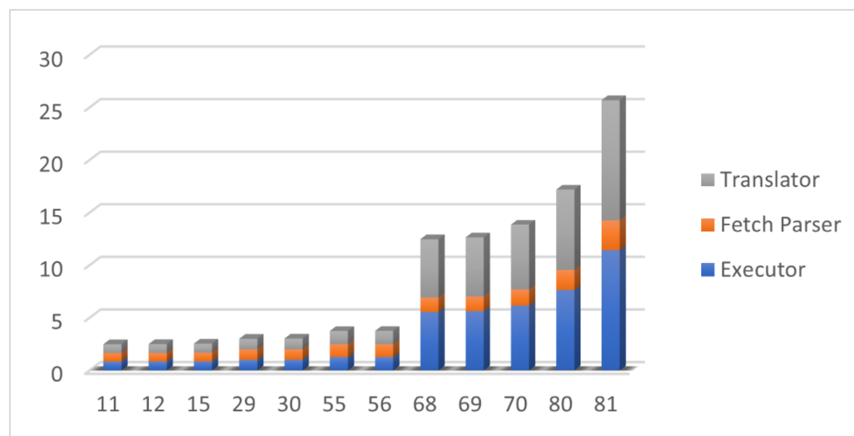


Figure. 6.19: PG 4 - Parser, Translator and Executor Times

In Cassandra, the "OR" operator is not supported. Furthermore, the prototype is unable to generate a well-formed query for Cassandra if the search filter is applied to multiple fields producing an error, irrespective if the "OR" or "AND" operator was used. MongoDB and Neo4j that consistently produced results correctly and without errors. Although, a number errors were reported, the prototype as minimum expectation correctly released the resources resulting in lower memory and CPU footprint.

6.3.5. Apply a limit to the dataset retrieval process

The next test cases dealt with data restrictions enforced by the participants. The Redis target was intentionally ignored as it does not support this feature. The other storage targets were able to restrict the rows based on the specified unified query without errors. The performance was satisfactory, producing effective results in a timely manner, highlighting no errors.

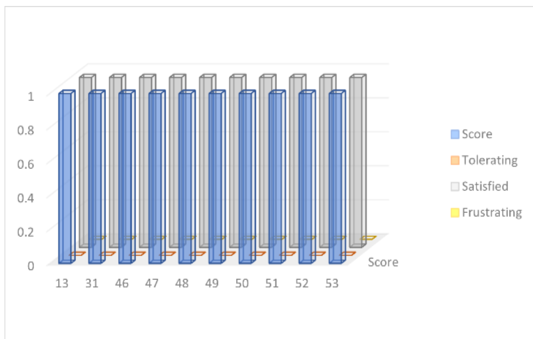


Figure. 6.20: PG 5 – Apex Scores

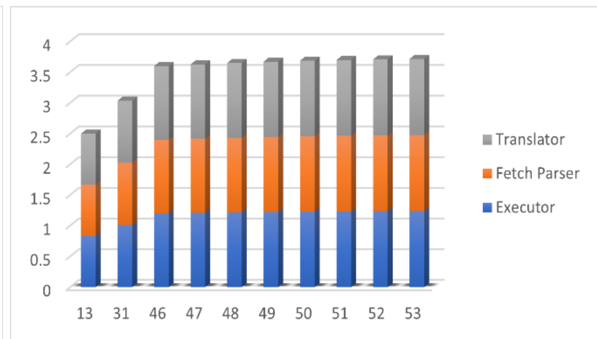


Figure. 6.21: PG 5 - Processing Times

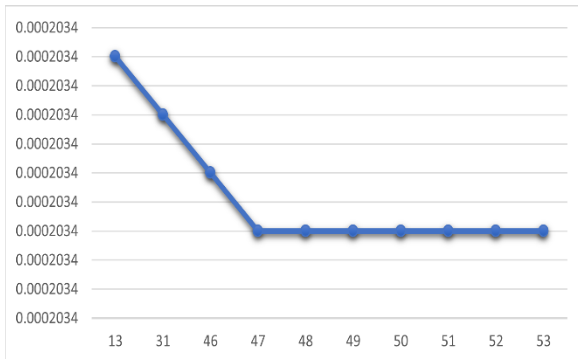


Figure. 6.22: PG 5 - CPU Processing Times

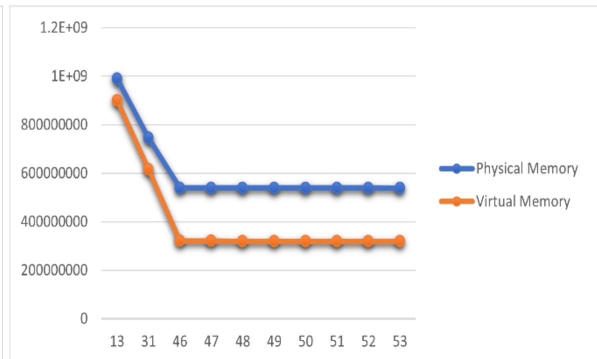


Figure. 6.23: PG - Memory Allocations

6.3.6. Apply sorting to the dataset retrieval process

Participants were tasked to arrange data in an expressive way. The prototype provided a sorting mechanism which enabled participants to examine data in a meaningful way. The overall performance of the automated tests operated within the bounds of the expected outputs.

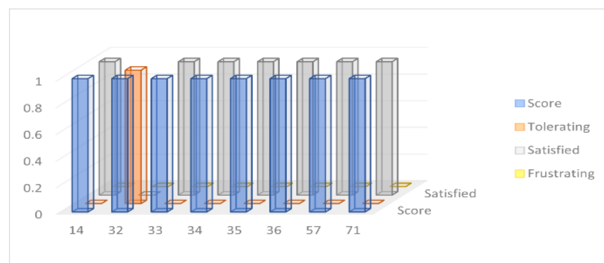


Figure. 6.24: PG 6 - Apdex Scores

Test case 32 was the only test that performed tolerable due to the volume of data that required sorting, as shown in Figure 6.24. While participants were not specifically tasked to perform search initiatives on the data, the idea was to determine how well different features co-operate. This resulted in errors as discussed in section 6.3.3 and section 6.3.4.

Figure. 6.25: PG 6 - Errors

Test case 14 outright failed whereas test case 71 resulted in a partial failure. In the instance of test case 71, the prototype was able sort the fields for the other specified target storage models to match the participants intent. The prototypes inability to successfully generate a well-defined Cassandra query with filters was the primary reason for the partial failure.

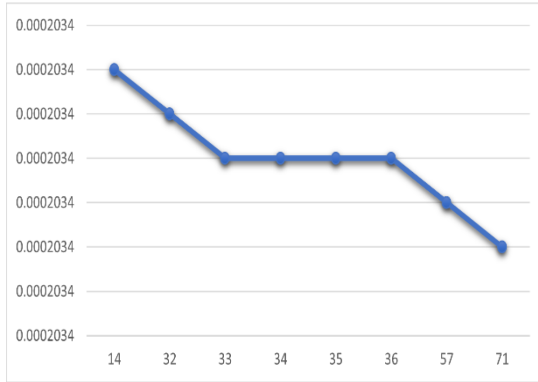


Figure. 6.26: PG 6 - CPU Processing Times

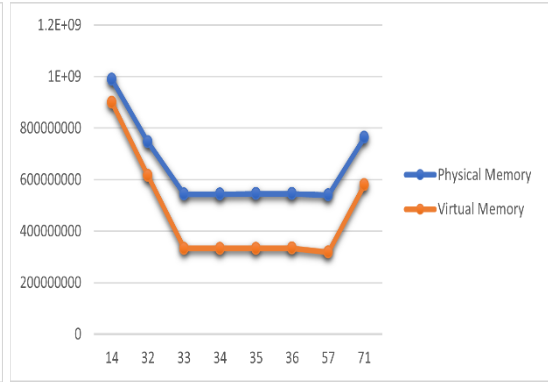


Figure. 6.27: PG 6 - Memory Allocations

Test case 71 included the Redis storage option as a target in the unified query; however the intent of the query does not match the result. This caused the memory allocations to spike as the prototype performed additional tasks it was not expected to perform, Figure 6.27. This impact was not significant in this case as the Apdex was still within the acceptable bounds. It should be noted that the prototype does not explicitly cater for Redis sorting as it is not supported natively. This is not a use case normally associated with Redis; the results return are incidentally, Thus the Redis query process was not reported as shown in Figure 6.28.

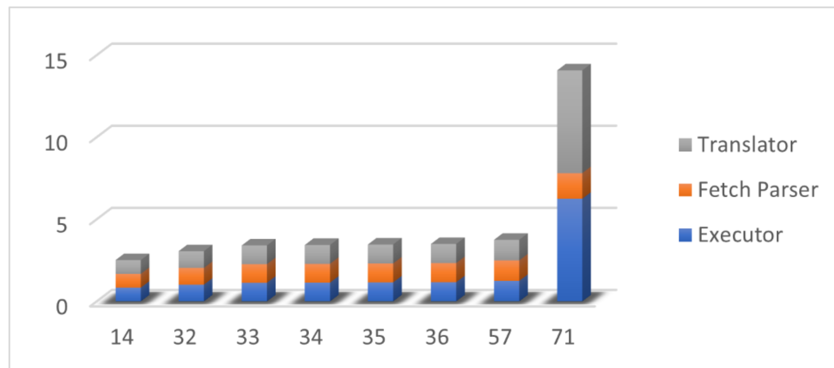


Figure. 6.28: PG 6 - Parser, Translator and Executor Times

6.3.7. Aggregation on a datasets

The aggregation performed by means of the unified query intent through simulations indicated the translation mechanism was able to handle complex queries in with the optimal timeframe. The Apdex scores reported no frustrating results considering how the prototype applied unique intricate algorithms specifically for Cassandra, MongoDB and Neo4j.

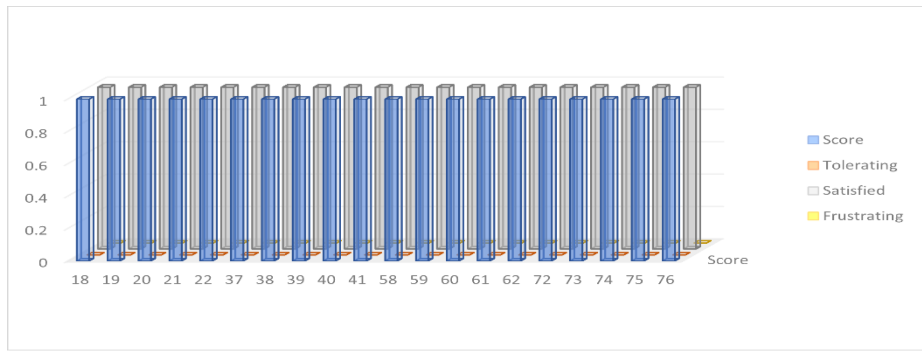


Figure. 6.29: PG 7 - Apdex Scores

Each of the forementioned storage models interprets the unified query in a special manner to ensure the native is well-defined. The participants performed a number of AVERAGE, COUNT, MAX, MIN and SUM actions (see Table 4.3) as part of the aggregation tests. The target model, Neo4j, was the worst performing storage option relative to MongoDB and Cassandra as indicated by Figure 6.30. This was influenced by the relations due to the complex relations that exist within this schema model. The prototype mapping and discovery mechanism for Neo4j informed by the metamodel repository, is more intricate than the other supported storage models. For MongoDB, Cassandra and Redis can view as more 'relational', i.e. tabular. Neoj4 on the other hand, required additional processing logic to firstly ensure the correct nodes where gathered even though it may not be explicitly specified in the participant's unified query intent. Finally based on the query intent, the optimal vertices and edges (i.e. relationship) had to be retrieved in order for the native query to return the correct result.

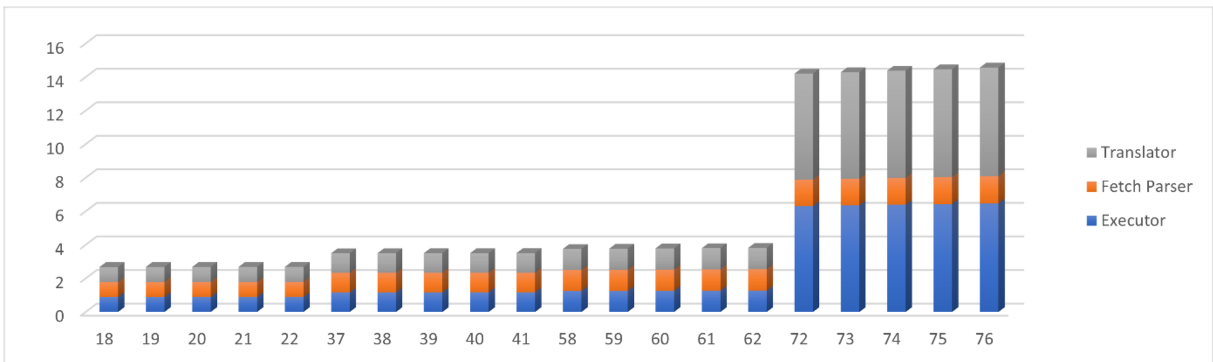


Figure. 6.30: PG 7 - Parser, Translator and Executor Times

The researcher observed when aggregated functions were specified on a field which does not exist in the underlying storage model, the prototype created a native query without completely fulfilling the user intent. As in the occurrence of test cases 72 to 76, the MongoDB storage does not contain data regarding student transcripts. This ultimately produced an aggregated result without the intended fields which obfuscated the output, leading to misrepresentation of the actual data. Figure 6.32, illustrates this fact as the prototype produced more data than expected resulting in an increase memory allocations.

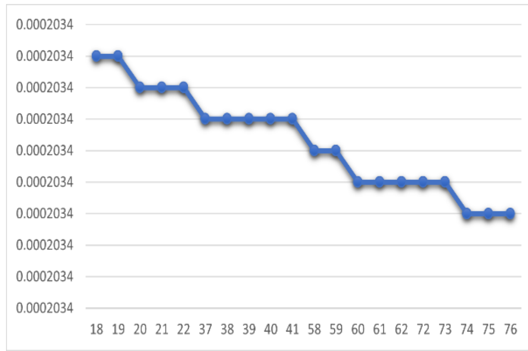


Figure. 6.31: PG 7 - CPU Processing Times

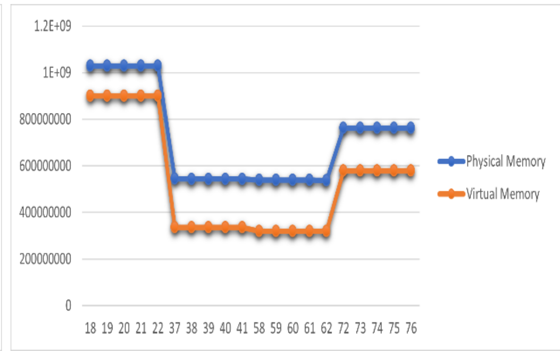


Figure. 6.32: PG 7 - Memory Allocations

6.3.8. Update existing dataset

The prototype only supports rudimentary updates. As shown in Table 6.33, the performance of the modification intent indicates no prolonged time allocated to the query processing. All updates specified by participants, included a filter which contributed to the optimal results shown in the apdex scores.

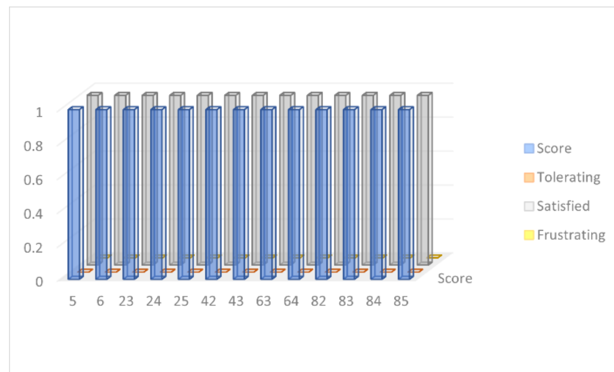


Figure. 6.33: PG 8 - Apdex Scores

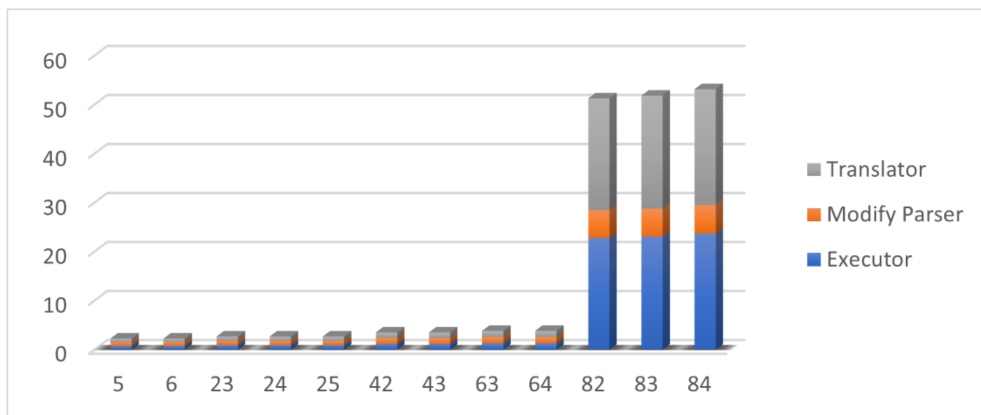


Figure. 6.34: PG 8 - Parser, Translator and Executor Times

The researcher observed specifying a date value in the “Modify” intent resulted in an error and produced an unexpected result whereby no native queries were generated. The researcher determined that the prototype’s parsing mechanism was unable to parse date values (test case 85).

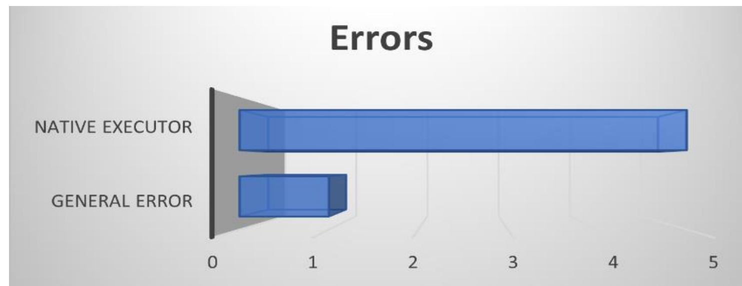


Figure. 6.35: PG 8 - Modification Errors

The Redis storage model does not explicitly support an update function, therefore the prototype firstly gets the record then applies the update in memory then execute the "set" command on Redis to invoke an update. Test case 5 and 6 in Figure 6.36 and Figure 6.37 illustrates the increase in resources when the intent was actioned.

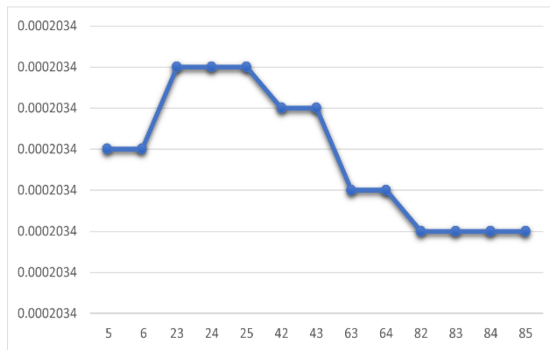


Figure. 6.36: PG 8 - CPU Processing Times

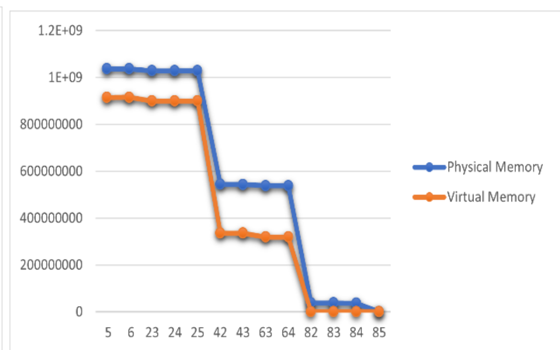
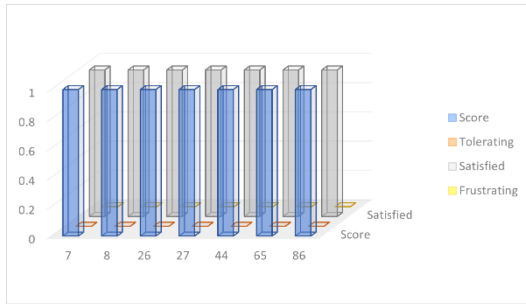


Figure. 6.37: PG 8 - Memory Allocations

As emphasised in 6.3.3 and 6.3.4, the failures detected in test case 24, 25, 82 and 83 relates to the filter not being specified in the primary index for Cassandra. While the "*Fetch*" intent in the instance of Cassandra does allow for search filters to be places on non-indexed fields, for the "*Modify*" intent the search filter on updates are only permitted on the primary index. Targeting multiple data stores for modifications the prototype shown an increase in the amount of time taken which was expected (Figure 6.34) but amount of memory allocate to the process decreased significantly. Initially, the researcher concluded that the modification algorithm required a smaller memory a footprint when compared to "*Fetch*" intents. However the decrease in resources utilised correlates with errors occurring.

6.3.9. Data inserts

The final participation group perform simple data inserts. The results obtained were similar to the participation group in section 6.3.8.



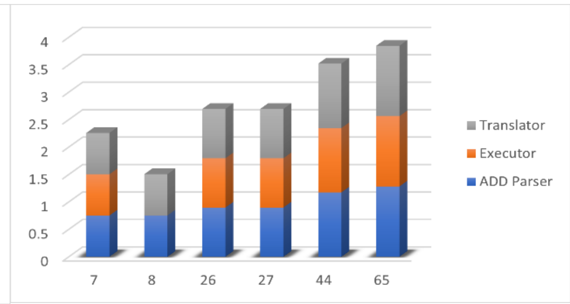


Figure. 6.38: PG 8 - Apdex Scores **Figure. 6.39**: PG 8 - Execution Times

The Redis and Cassandra data storage models requires the primary keys to be present in the "*Add*" intent. This was proven in the test cases 8 and 27 where the native generated queries failed to adhere to the respective storage models principles. The parser in test case 86 also failed due to the date format. As with the "Modify" intent, the native queries was not generated for any of the targeted storage models.

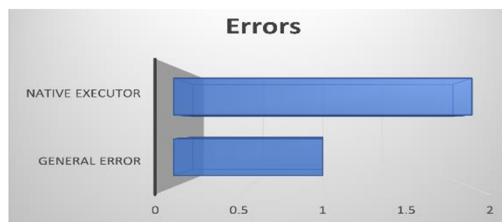
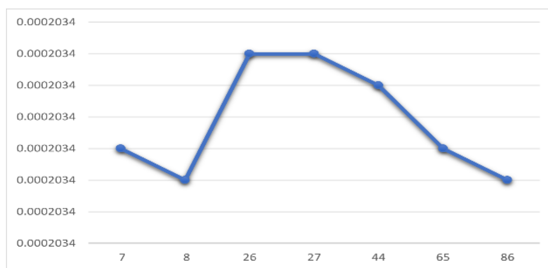


Figure. 6.40: PG 9 - Errors

The researcher observed that no new discoveries were made after analysing the data besides the fact that a the primary key of the underlying schemas must be present in the intent. When targeting multiple storage models, there were instances where the specified fields that was not utilized by each of the supported storage options i.e. not defined in the schema. As indicated, below that did not have a significant impact on the overall performance.



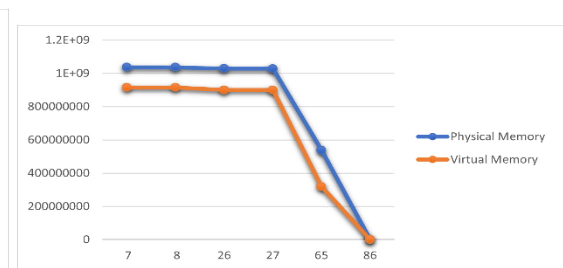


Figure. 6.41: PG 9 - CPU Processing Times **Figure. 6.42**: PG 9 - Memory Allocations

6.4 Significance of Results

The significance of the prototype results was attained through generalized estimating equations (GEE). The statistical design choice is an extension of general linear model (GLM), as the necessary observations, i.e. simulated tests conducted, entailed group of correlated data produced by the prototype. The study's final objective, *RO4*, was to determine how effective and efficient the prototype handled the various simulated queries. Therefore the observations was examined from two perspectives; the overall response times of the simulated tests and the amount of resources consumed during the execution.

Apdex Case Summary

In the instance of evaluating the response times of the prototype, the Apdex score was taken into account as it provides the perceived satisfaction levels of the end users' experience interacting with the unified query platform. A binomial and multinomial distribution method was applied to the Apdex to provide insights into the relationships between the correlated parameters and the final outcome. In the case of the GEE relating to the Apdex, two models were procured. The first model treated the "Frustrating" and "Tolerant" category as the response and the "Satisfied" category as the reference, hence a binomial distribution method with a logit link function was utilised. The second model treated the three categories as separate outcomes which is closer to how the prototype represents these results. Hence a multinominal method was chosen with a cumulative logit. In addition, this model produced a more consistent significant predictors and was be more interpretable due to the threshold structure as described in 6.2.

Table 6.2: Apdex : Model Description

Description	Value
Probability Distribution	Multinomial
Link Function	Cumulative logit
Working Correlation Matrix Structure	Independent
Degrees of Freedom	1
Dependant variable	ApDexFrustrating_tolerating_satisfied
Number of observation	256
Scale	1
Threshold	
Frustrating	1
Tolerating	2
Satisfied	3

Table 6.3: Apdex : Covariates

Description	N	Minimum	Maximum	Mean	Std. Deviation
Max	256	0.0078	56.28	0.9701	4.8445
Min	256	0.0036	56.28	0.967	4.8451
StdDev	256	0	0.0312	0.0013	0.0036
Sum	256	0.0078	56.28	0.9725	4.844
Fifteen Minute Rate	256	10.979	47.734	16.521	12.516
Five Minute Rate	256	9.1911	47.207	15.01	11.934
Mean Rate	256	0.6999	23.638	2.4598	3.8206
One Minute Rate	256	3.1632	44.162	8.7514	9.2592

The model answers the probable success outcomes relating to effectiveness of the prototype. Table 6.3. shows the covariates calculated which was subsequently used in conjunction with the Wald Chi-Square to measure the significance predictors. In short, the model was design to determine the likelihood the simulated tests will result in the "Frustrated" and "Tolerating" category. This speaks directly to the user experience. The data indicates that in both cases, if

the individual response times of the each component (parser, translator and executor) take longer than expected, the user is likely to experience a frustrating or tolerating outcome.

Table 6.4: Apdex : Generalized Linear Model

Parameter	B	Std. Error	95% Wald Confidence Interval		Hypothesis			95% Wald Confidence Interval for Exp(B)	
			Lower	Upper	Wald Chi-Square	Significance	Exp(B)	Lower	Upper
Frustrating	-4.46	0.9387	-6.299	-2.62	22.569	<0.001	0.012	0.002	0.073
Tolerating	-2.518	0.8762	-4.235	-0.8	8.257	0.004	0.081	0.014	0.449
Max	-1304.257	882.9744	-3034.855	426.341	2.182	0.14	<.001	0	1.44E+185
Min	1529.621	916.9917	-267.649	3326.892	2.783	0.095	.[a]	5.77E-117	.[a]
StdDev	3741.393	2152.4454	-477.323	7960.108	3.021	0.082	.[a]	5.03E-208	.[a]
Sum	-225.38	91.0728	-403.879	-46.881	6.124	0.013	<.001	3.96E-176	4.37E-21
Fifteen Minute Rate	-13.155	3.7252	-20.457	-5.854	12.472	<0.001	<.001	1.31E-09	0.003
Five Minute Rate	17.044	4.8689	7.501	26.587	12.254	<0.001	25235016.45	1809.495	3.51925E+11
Mean Rate	1.786	0.6559	0.5	3.071	7.414	0.006	5.965	1.649	21.571
One Minute Rate	-4.874	1.5008	-7.815	-1.932	10.545	0.001	0.008	0	0.145

** .[a] - set to system missing due to overflow*

The threshold parameter for the category "Frustrating" is -4.46. The odds of an observation falling into the "Frustrating" category increases by approximately 1.2% for each one-unit increase in the predictor variable. The threshold parameter for the category "Tolerating" is -2.518. The odds of an observation falling into the "Tolerating" category increase by approximately 8.1% for each one-unit increase in the predictor variable. Hence, probabilities favour a more tolerant and by default a more satisfied user experience. The thresholds relates to the continuous predictor variables, listed as parameters in the model. The thresholds, in combination with these predictors, determine the transition points between the three satisfaction categories. The data with a smaller p-value (< 0.05) indicates that the parameters are statistically significant. The Exp(B) values provide insight into the direction and magnitude of the effect for each variable. For the mean and five minute rate, the data suggests that a positive coefficient implies a positive association with the dependent variable. It indicates that odds of an observation falling into a specific category increase as the independent increases. This often happens when dealing with extremely large or small values. Based on data, hypothesis and ultimately significance, the model indicates that it is significant enough where the prototype's unified query has a high probability of being effective.

CPU Case Summary

The CPU was evaluated to determine if the prototype recklessly consumed the physical machine's resources during the simulated tests. The statistical analysis was inspected from three viewpoints; the CPU physical memory, private memory and utilisation. The data was subjected to descriptive statistics providing a concise summary of the main features of a dataset; making data more understandable, meaningful, and interpretable. This entails characteristics

such as the variability, the shape and distribution and frequency of data in question. This afforded the researcher, the foundation for further statistical analyses and help in making informed decisions based on data. The gamma distribution was selected, shown in Table 6.6 and 6.8, as it is versatile and can represent a range of shapes, from exponential to normal-like distributions, depending on its parameters. The flexibility gamma offers enables the researcher to accommodate different shaped data patterns.

The descriptive analysis performed on the CPU utilisation suggest a consistently low and stable level of CPU usage. The usage operated at 0.0203% during the simulations. The prototype data shows an extremely low mean and variance suggesting that the CPU utilization is very stable and consistently low across all the observed cases (Table 6.5). This could also indicate that the system was not under heavy computational load at the time of the simulations or in the extreme case that more varying test was required. The varying responses was close to the mean value therefore suggesting the inner workings of the prototypes components are consistent and stable. However, the normality tests indicate that the data doesn't follow a normal distribution, which is expected for CPU utilisation data. Further analysis, considering the context of the system and objective, would provide a more complete understanding of the system's efficiency and CPU utilization.

Table 6.5: CPU Utilisation : Statistical Descriptives

		Statistic	Std. Error
	Mean	0.000203	0.0000000
95% Confidence Interval for Mean	Lower Bound	0.000203	
	Upper Bound	0.000203	
	5% Trimmed Mean	0.000203	
	Median	0.000203	
	Variance	0.000	
	Std. Deviation	0.0000000	
	Minimum	0.0002	
	Maximum	0.0002	
	Range	0.0000	
	Interquartile Range	0.0000	
	Skewness	-0.056	0.152
	Kurtosis	-1.069	0.303

The data on physical memory indicates significant variability, non-normality, and a skewed distribution. The choice of the gamma distribution in the GLM reflects an attempt to model these characteristics. The statistics reveal that physical memory values vary widely, with a substantial range and high standard deviation. The negative skewness suggests that there might be a concentration of lower values, and the positive kurtosis indicates heavier tails in the distribution. The non-normality tests reinforce that the data does not follow a normal distribution, which is expected for physical memory values in many systems.

Table 6.5: CPU Physical Memory : Model Description

Description	Value

Probability Distribution	Gamma
Link Function	Log
Working Correlation Matrix Structure	Independent
Degrees of Freedom	1
Dependant variable	CPU_Physical_Memory
Number of observation	256

Table 6.6: CPU Physical Memory : Dependant Variable and Covariates

	Description	N	Minimum	Maximum	Mean	Std. Deviation
Dependant	CPU_Physical_Memory	253	4096.0000	1123409920.0000	722200810.750988	256289565.7031443
Covariates	Max	253	0.0078	56.2799	0.835374	4.6131766
	Min	253	0.0036	56.2799	0.832257	4.6137301
	StdDev	253	0.0000	0.0312	0.001323	0.0036085
	Sum	253	0.0078	56.2799	0.837824	4.6127493
	Fifteen Minute Rate	253	10.9794	47.7341	16.575332	12.5797433
	Five Minute Rate	253	9.1911	47.2066	15.047585	12.0001477
	Mean Rate	253	0.6999	23.6383	2.407274	3.8123871
	One Minute Rate	253	3.1632	44.1621	8.724255	9.3107710

Table 6.7. documents the intercept parameter which serves as the baseline value for the physical memory when everything else is zero. It is significantly different from zero, meaning even when other factors are zero, there's still some memory being used. This would account for other processes running concurrently when the prototype was subjected to the simulated tests. When analysing the different rates, the data shows that changes in the “FifteenMinuteRate”, “FiveMinuteRate”, “MeanRate” and “OneMinuteRate” parameters have a significant impact on physical memory. When these rates go up or down, physical memory usage tends to change. The other parameters in the model indicated no impact on the physical memory.

Table 6.7: CPU Physical Memory : Generalized Linear Model

Parameter	B	Std. Error	95% Wald Confidence Interval		Hypothesis			95% Wald Confidence Interval for Exp(B)	
			Lower	Upper	Wald Chi-Square	Significance	Exp(B)	Lower	Upper
Intercept	20.074	.0621	19.953	20.196	104425.659	<0.001	522598681.314	462690895.641	590263141.732
Max	15.689	87.7493	-156.296	187.675	0.032	0.858	6514094.855	1.323E-68	3.207E+81
Min	-21.324	89.8984	-197.522	154.874	0.056	0.813	<0.001	1.650E-86	1.823E+67
StdDev	-44.786	207.9218	-452.305	362.733	0.046	0.829	<0.001	3.685E-197	3.413E+157
Sum	5.621	8.0305	-10.118	21.361	0.490	0.484	276.247	4.034E-5	1891872346.476

Fifteen Minute Rate	4.839	0.2902	4.270	5.407	278.053	<0.001	126.288	71.510	223.026
Five Minute Rate	-6.200	0.3814	-6.947	-5.452	264.293	<0.001	0.002	0.001	0.004
Mean Rate	-0.781	0.0520	-0.883	-0.679	225.684	<0.001	0.458	0.414	0.507
One Minute Rate	1.741	0.1204	1.505	1.976	209.141	<0.001	5.701	4.503	7.217

The CPU private model followed the same approach as the physical memory model as shown in Table 6.8. In simple terms, the regression model attempted to discover how much private memory was consumed by the prototype. However, there is room for further refinement of the model, as the distribution is slightly positively skewed and has a negative kurtosis, indicating a relatively flat distribution. It's crucial to interpret these results in the context of the prototype system and consider what the practical implications are for determining the efficiency of its resource use.

Table 6.8: CPU Private Memory : Model

Description	Value
Probability Distribution	Gamma
Link Function	Log
Working Correlation Matrix Structure	Independent
Degrees of Freedom	1
Dependant variable	CPU_Private_Memory
Number of observation	256

Table 6.9: CPU Private Memory : Dependant Variable and Covariates

	Description	N	Minimum	Maximum	Mean	Std. Deviation
Dependant	CPU_Private_Memory	253	65536.0000	952733696.0000	547800934.197629	280354780.3088633
Covariates	Max	253	0.0078	56.2799	0.835374	4.6131766
	Min	253	0.0036	56.2799	0.832257	4.6137301
	StdDev	253	0.0000	0.0312	0.001323	0.0036085
	Sum	253	0.0078	56.2799	0.837824	4.6127493
	Fifteen Minute Rate	253	10.9794	47.7341	16.575332	12.5797433
	Five Minute Rate	253	9.1911	47.2066	15.047585	12.0001477
	Mean Rate	253	0.6999	23.6383	2.407274	3.8123871
	One Minute Rate	253	3.1632	44.1621	8.724255	9.3107710

As part of the statistical test, goodness of fit model was applied to the data to determine whether a set of observed values match those expected under the GLM. The test highlighted that the model may need improvement to better capture the patterns in private memory utilization. The Quasi Likelihood under Independence Model Criterion (QIC) produced a value of 575.823 while the Corrected QIC produced a value 85.409. In this case, the model may need improvement to

better capture the patterns in private memory utilization. In certain instances, there were factors had no impact, while others, such as the “FifteenMinuteRate”, “FiveMinuteRate”, “MeanRate” and “OneMinuteRate” parameters, were significant enough in predicting changes in private memory usage. This pattern correlates to the discoveries made in the usage of physical memory.

Table 6.10: CPU Private Memory : Generalized Linear Model

Parameter	B	Std. Error	95% Wald Confidence Interval		Hypothesis			95% Wald Confidence Interval for Exp(B)	
			Lower	Upper	Wald Chi-Square	Significance	Exp(B)	Lower	Upper
Intercept	19.907	0.0980	19.715	20.099	41223.558	<0.001	442182966.444	364874493.748	535871317.846
Max	161.251	138.4983	-110.201	432.702	1.356	0.244	1.072E+70	1.381E-48	8.322E+187
Min	-206.616	141.8904	-484.716	71.484	2.120	0.145	<0.001	3.095E-211	1109992398426045400000000000000000.000
StdDev	-474.907	328.1716	-1118.111	168.298	2.094	0.148	<0.001	0.000	1.232E+73
Sum	45.345	12.6749	20.503	70.188	12.799	<0.001	49335573067586990000.000	802161403.534	3034300527529010600000000000000.000
Fifteen Minute Rate	9.010	0.4580	8.112	9.908	387.038	<0.001	8185.440	3335.820	20085.447
Five Minute Rate	-11.661	.6019	-12.841	-10.482	375.327	<0.001	<.001	2.649E-6	2.805E-5
Mean Rate	-1.853	0.0821	-2.014	-1.692	509.983	<0.001	.157	0.133	0.184
One Minute Rate	3.482	0.1900	3.109	3.854	335.932	<0.001	32.518	22.409	47.188

In summation, the data model for the Apex revealed correlations with significant relationships among the metrics, indicating that certain aspects of the query prototype, such as response times and rates, were interrelated. Some of the parameter estimates have been set to system missing due to overflow, indicating that the calculated values were too large to be represented in the output which may require further analysis. However, since the timer-related metrics suggest that there is a notable relationship between these metrics and user satisfaction. This implies that the Apdex, which incorporates these metrics to measure the overall user experience, is responsive to variations in system performance. Thus concluding that the prototype probability of the prototypes effectiveness has been realised.

The CPU statistical data varies in terms the researcher supposition that the prototype is able to efficiently utilise the machines resources. The normality tests (Kolmogorov-Smirnova and Shapiro-Wilk) indicate that CPU utilization does not follow a normal distribution. Hence a model

could not be constructed as the related data was insufficient. Therefore this study cannot definitively conclude the efficient use of the CPU utilisation. The data does show low and stable CPU utilization which is generally considered efficient, as it indicates that the system is not overburdened. However, further analysis and possibly additional simulations might be needed to understand the specifics of the system's workload and whether this low utilization is optimal. The CPU physical and private memory on the other hand suggests that the patterns in timers and rates correlates with the memory consumption. These indicators demonstrate a wide range of memory consumption whereby the researcher concluded that sufficient memory was consumed and release by the CPU. This speaks to the general efficiency of the prototype. At the same time the wide range of variability suggests optimizing the system based on these predictors might enhance overall efficiency.

6.5 Context for the Prototype’s Findings

The significance of the results outlines a number of revealing straits in evaluating the efficacy of the prototype. The comprehensive statistical methodology applied to the result set from the automated tests uncovered how well the prototypes performed its designed capabilities under varying scenarios. The testing focused on the prototype’s ability to create executable queries along with its error handling capabilities, resource consumption and storage model compatibility. This precipitated the researcher to analyse and dissect its performance metrics relating to its efficacy in error handling, filtering capabilities, aggregation functions, data mapping and manipulation outputs across different storage targets.

The performance varied across the targeted storage models, with each exhibiting strengths and weaknesses. The storage system, Neo4j performed relatively worse when compared to the other targeted storage models. However, this was highly dependent on the extent of the intricate relationships between nodes. This discovery essentially highlighted areas for improvement in the Neo4j storage’s translation component. MongoDB and Cassandra showed a more reliable and stable performance, even with complex schemas, indicating that applied transformative algorithms were efficient. Redis on the other hand consumed more memory due to its inability to compress data efficiently. This is in part due to the prototypes default query mechanism whereby the prototype will attempt to produce a query regardless of the intent, as articulated in section 6.6.3. In most cases, the data analysed demonstrated that the prototype chosen the optimal path execution path, as indicated in section 6.4, in the Apdex Case Summary.

The prototype generally handled errors well, however in certain cases there were instances where the error messages are inaccurate leading to unexpected load on machine’s resources. This behaviour caused the incorrect error messages to be rendered deviating from the intended logging process. This is especially true when the unified queries targeted Redis and Cassandra storage models. Errors were also generated by the prototype, especially in cases where participants applied the filtering mechanism, i.e. "AND/OR" operators. Cassandra specifically raised errors in generating incompatible native queries when applying search filters. In the case of Redis, where such operations were not supported, the prototype returned incorrect results.

This is primarily due to compatibility issues observed by the researcher, particularly with unsupported operations or incorrectly mapped fields in the metamodel pertaining to certain storage models.

Throughout the experiment, a recurring theme emerged, leading to two distinct reflections. Firstly, instances where the prototype yielded unfavourable outcomes or exhibited unexpected behaviour. Secondly, on the occasions when errors occurred, either due to query syntax and semantic faults or due to a mismatch in query intent. These factors resulted in the misuse of the machine's resources whereby either where more memory was utilised or the prototype held onto CPU for a longer time period. Thus allowing the prototype to generate an inefficient query path. Thereby making a strong case to improve areas within the prototype such as the data and feature mapping within the metamodel as well as the translation and execution mechanism leading to efficient query generation.

As pointed out in section 2.8 and observed in section 6.4, there is an immediate performance penalty when the prototype is executing the native queries simultaneously. To mitigate this in future, the solution may require additional infrastructure such as more CPU cores and memory; quite possibly improved networking speed, should the prototype be deployed on a client-server type Architectural model. Data quality could also be a concern should the translation process via the *Modify* and *Add* intents not be managed appropriately. In spite of all of these factors the prototype provided improved data accessibility and efficiency. Overall, the prototype demonstrated several positive attributes, including robust evaluation methodologies, adaptable performance across storage models, reliable operation, and satisfactory error handling. Considering, that in order to access the data without the prototype, multiple systems or tools would be required to interrogate the individual NoSQL databases, increasing the time to consolidate and analyse the data. Thus, the prototype abstracts away the complexities of each NoSQL database in this study enabling data to be processed into information at a faster rate. This facilitates consumers to make informed decisions effectively and efficiently, i.e. real-time insights despite the initial performance trade-off. Additionally, the evaluation process provided valuable insights into areas for further refinement and improvement, paving the way for future enhancements to optimize performance and usability. Considering all factors, the prototype's demonstrated a high probability of efficacy.

6.6 Implications of Findings

The implications of the findings discussed provides valuable insights into the approach taken to develop the prototype for a unified platform based on the results attained in section 6.3. The findings played an important role in evaluating the performance and overall effectiveness of the prototype in relation to its objectives. The observations made will assist in further improvements and guide the decision making for future research endeavours in unified query system domains. Although the results indicated that a number of shortcomings, this study concludes that the prototype demonstrated its effectiveness and efficiency. This was achieve through the prototypes ethos, the level of abstraction it adhered to, the results returned based on the

participants intents, the query processing, the error handling approach and finally the performance.

6.6.1. Prototype's Ethos

Firstly, the goal of the developed unified query platform using the polyglot approach was to be agnostic and interoperable. This ethos was applied to the prototype to suppressed the complexities of the supported physical NoSQL storage systems from the users (Kolonko & Müllenbach, 2020). Essentially the prototype in this study served as a spokesperson for the unified query directing requests to the native databases through its supported protocols; relaying the results back to the callers or users. This was achieved by using a natural language with coherent lexical, syntactical and sematic paradigms represented as the unified query (Yang, Zhang & Tong, 2022). The text-based language, even though it was not specifically tested in Chapter 5, the study makes the assertion that humans are able to understand the query language. While this requires additional efforts in future research, the aim is that the natural language should make learning the elements of the unified query easier, resulting in greater adoption.

6.6.2. Prototype Abstraction

An abstract unified data model was implemented that served as a delegate to the underlying disparate native storage models. This entailed a comprehensive representation of each native data schemas that promoted accessibility (Kolonko & Müllenbach, 2020). The unified data model had to be query able; clearly separating the structural complexities of each native storage data models. A metamodel was utilised which uniformly catalogued the abstract and native schemas was the fundamental concept that serves as the first step for data exchange between the abstract and physical models. It served as the glue aimed to bridge the relationship gap between abstract and source. It indexed the data fields, attributes and schemas or ontology of both the unified and native data models linking the properties of the abstract schema to the physical native schemas (Hewasinghage et al., 2021). Without the metamodel, the prototype's objectives would not have come to fruition.

6.6.3. Query Intents of the Prototype

The prototype provided a clear and actionable directive that matched the intent of a given instruction by the participants. By extension, this entailed converting the natural language into an abstract syntax tree that captures the data fields, attributes and schemas (Zhang, 2020). The abstract syntax tree was superimposed on natural language, encapsulating set a mechanical instructions for the middleware to execute. While the majority of automated tests produced results that match the intent, in certain instances especially with the Redis target model, the user received an invalid result set.

A common theme to transpired, relates to the fact that if the prototype is unable to exactly match the users instructions, it either tries to offer the "next best" result or defaults to it a basic intent. This occurs without any indication to the user or participant that the query does not matched the

user's request. Alternatively, the researcher had to analysed the results to determine if the intent matches the dataset returned. This occurred in a few instances especially where either the data field was not supported natively or the feature was not supported by the underlying storage model.

6.6.4. Query Processing

The goal was to interrogate the entire schema of the underlying native storage models and return the associated dataset. The unified query system isolated certain key features of the system, such as query parser, query translators, data retrieval mechanisms and output bindings. This modularised approach promoted scalability, flexibility and maintainability (El Maghawry & Dawood, 2010). A strategy that was not explicitly expressed, related to the ability for components to discover each other at the runtime. Selecting the appropriate design pattern reflected this in the most apt way as the intent of the user, consequently built a programmatic call stack that matched the query directive (Gahlyan & Singh, 2018). The suitable features of each component was matched together in a linear way which form the query executable path.

The unified query platform for the prototype naturally required a translator accountable for converting the abstract query represented by the AST into suitable native queries. Each targeted model implemented its own mechanisms for accomplishing this. The translation components of each targeted model may have shared certain constructs but how it used those constructs was unique to the respective storage models. The objective was to generate desirable and well-defined queries over the target models that was required to be executable. In order to reduce the memory and CPU footprint on the application, the researcher took the approach of delegating most of the query execution labour to the supported NoSQL DBMS. The overall performance in the results reflect this as not a single automated test scored an Apdex of zero.

This approach serve well, however the prototype does not make any definitive mitigation for features that are not supported natively, which result in adverse output. The query executor does not perform a native parsing check before it attempts to run the generated query. It relies on the translator as 'all knowing'. This gap was made notably visible by the automated tests, given the errors and invalid results obtained from Cassandra, Redis and MongoDB. In certain instances as discussed in section 5.5, either the featured intent was not incorporated in the native query or the translator generated a native query that was destined to fail and finally a completely different result to the one expected.

The significance of the results suggests that further analysis is required to determine the efficacy of the prototype. This implies that more tests are required in order to reached an unbiased conclusion. Based on the probable outcomes for this study, the data leans more towards suggesting that the prototype is indeed is effective and efficient. While the current data patterns suggest that more analysis may be required, the probabilities does not unequivocally advocate that the prototype is not fit for use. On this basis, the researchers concludes that the prototype operates in a manner that is fit for purpose.

6.6.5. Prototype Error Handling

The prototype ideally segregated key features in a such a way that certain parts could remain operational even in adverse circumstances. In short the unified system was fault-tolerant in order to gracefully handle failures. Errors should be clearly communicated to users if encountered during the query process. Since the prototype compromised of a number of components, it was important to provide context as to which part of the system has manifested the error. The prototype in this regard needs a lot more attention as the result shown that the error messages encounter, while it provide context, it failed to render an actionable messages.

6.7 Contributions to Knowledge

This study contributes to the existing body of knowledge in under two different circumstances. Firstly, it contributes to the theory of information systems as it used DSR as a methodology to gain understanding of the problem domain (Hevner et al., 2004). Secondly, it contributes to practical procedures encompassed through the developed prototype.

6.7.1. Contributions to Theory

The study applied the DSR method to guide the research endeavour with aim of evaluating how well a unified query platform prototype may be developed effectively and efficiently. It used the DSR process model to provide the necessary research rigour removing any inane actions, carefully detailing how the research activities relates to the study problem (Vaishnavi, Kuechler & Petter, 2019). Moreover, it operated under the guidelines set out by Hevner et al. (2004), to create a purposeful artifact through scientific means and enhance the existing information systems body of knowledge.

6.7.2. Contributions to Practices

The practical contributions for the research study is embodied in the proposed prototype's data and physical implementation to address the lack of a guidelines and practices around the development of unified query systems. The prototype consists of a range of design and architectural principles which will assist practitioners pursuing these types of systems. The data obtained through the automated tests should enables futures researchers to analyse this data and propose meaningful enhancements. This study provides valuable insights on the guidelines, design and architectural implementations exercised during the construction process of the artifact. The prototype attempts differentiate itself from similar solutions by amalgamating a set of established physical architectural principles in a unique way that firstly produces a working prototype; and subsequently a solution that promotes modification and easily is extensible.

6.8 Limitations

The study experienced the following limitations and challenges during the course of the research endeavour:

- The study initially proposed an automated schema identifier that's able to successfully affect the underlying native schemas through the prototype. Due to times constraints, this feature was excluded from the scope of the research project.
- The data utilised in the study was restrict to text-based data, thus excluding media type information such as images and videos.
- Any schemas updates were done manually, which opens the prototype to errors.
- The prototype was unable to handle complex data additions and updates, especially in the case of nested query processing based off existing data models.
- Updates could not be performed on complex fields within Cassandra database management system as it requires the entire object to be retrieved, update the identified field(s) then send the entire field back for modification.
- The study was limited to specific versions of the supported NoSQL data storage options. Any change in versions of the respective NoSQL database management system may render the solution obsolete or result in unified queries that previously operated successfully throw errors.
- The adaptors develop for the prototype relies a rudimentary security mechanism for the respective NoSQL database that needed connections to be authenticated.

6.9 Recommendations and Future Research

Based on the results obtained from the automated tests on the prototype the study proposes the following recommendations in the form of guidelines for developing a unified query platform. The guidelines are as follows:

- The unified query platform should start with a conceptual model, identifying the native schemas and properties then proceed to establish an unified data model that is able to represent the native schemas in a common way.
- The metamodel representing the unified and native schemas should be separated isolating any inconsistencies or errors to the individual storage catalogues.
- A natural language should serve as input to create an AST. A text-based language is the preferred method to serve as a unified query as its familiar to consumers of interrogating data and will most likely drive greater adoption.
- Compartmentalise the query intents, the query generators and query executable path with the appropriate design patterns. The researcher endeavour found that the chain of responsibility design pattern was deemed useful for the query intent, i.e. "*Fetch*", "*Modify*" and "*Add*". The query generators the applied the visitor design pattern where the elements of the AST was inspected and each element generated a part of the query. Finally the executable path for the query applied the strategy pattern which informed the prototype exactly which NoSQL targeted model to translate and execute.
- The design and architectural patterns applied must fundamentally advocate for the unified query platform to be easily extendable and adaptive to change, i.e. new storage data models should be easily added without having adverse effects on the existing integration.

- Decomposing the unified query platform into independent features enables parts of the system to be tested in isolation. By compartmentalising the features enables segments of the unified query platform to be built in such a way whereby a set of targeted systems tests around a single unified concept can be developed, improving the robustness and effectiveness of the solution.
- The error messages should be intuitive, guiding the user on how to resolve the issues that occurs during the query execution path.
- Infrastructure components must versioned and be viewed as immutable. With this in mind, the components should be viewed as disposable which is replaced when new versions are developed rather than updating an existing version. This serves the system two-fold, firstly it enables the system to be backwards compatible should the new versions of the implementation manifest adverse outputs. Secondly, it enables a degree of control when introducing new features to allow for a smooth transition between system updates.
- Since each of the native query results are wrapped in a format unique to each targeted databases drivers, a mapping mechanism is required that's able deserialize and map the results to the unified data model. A concrete class object is required to represent the unified data model.
- The unified platform's features should be explicitly mapped to the native features to provide some sort or indication to users, what functionality can be delegated via the abstract query to the individual target storage systems.
- A comprehensive monitoring system is required to provide ongoing awareness of how well the unified data platform is operating. The purpose of a clear reporting system is to proactively identify performance degradation or unforeseen errors that might occur.

Future works

There are a number of areas that requires improvements and further research. In particular, the metamodel in this study only catalogues the schemas along with the respective fields and attributes. It comprises a of collection of static files within this study's prototype. An ideal approach to defining a more dynamic metamodel which exists within a persisted repository. It should only be accessed through well-defined interfaces to provide a measure of control when maintaining or adding target systems. This will enable data to be accessed that may not have been in the original scope of work.

As indicated by this study, the data results in some of the test scenarios were unable to match the participants intent. Greater focus should be given to the unified query's feature mappings in relation to the native storage models capabilities. This will aid in defining an optimal query thus increasing the efficiency and more importantly returning the requested result. Defining the ideal constraints will aid in the development of optimized queries. In addition, native features that cannot be realized by the query generator, due the native storage option not supporting the use case, should be delegated to the middleware. The middleware should strive to enhance the

query execution path by include a decision engine that assert whether or not the expected features embedded in the unified query should be handled natively or via the middleware.

Future researchers should consider adding another level abstraction within the unified query domain. This abstraction should handle features that are not supported natively, thus providing complete and consistent traits to the unified query platform. This addition will of course require more resources, conversely it will enrich the user experience.

6.10 Summary

In this final chapter, the researcher reached the climax of the study, drawing the conclusion on the efficacy of the artifact. The reader's attention is drawn back to the primary research question with the aim of establishing how the research endeavour addressed the research problem through Design Science Research as a methodology. It highlighted the research rigor that was attained through the DSR process model and guidelines. By means of a systematic literature review under the stewardship of the circumscription process within the DSR process model, the research project was able to gather and identify knowledge from the relevant academic papers and existing solutions within the problem domain. The knowledge attained from past studies served to inform the decision-making process throughout the development of the prototype.

The researcher argues in favour of the prototypes utility based on the reporting metrics during a methodical evaluation. The finding indicates that the greater part of the protype's results demonstrated the desired outcomes thus fulfilling its objectives. The evaluation of the prototype proceeded to enhance existing knowledge on unified query platform initiatives. This effectively reached the philosophy of DSR as method for this study where a purposeful artifact was created. Furthermore the study encapsulates the learnings through a set of guidelines to aid in the development of unified query systems. It recognises the shortcomings throughout the study, especially those revealed in the evaluation of the prototype. The researcher brought to light the shortcomings of the solution and recommended how this may be addressed future research programs.

REFERENCES

Alharahsheh, H.H. & Pius, A., 2020. A review of key paradigms: Positivism VS interpretivism. *Global Academic Journal of Humanities and Social Sciences*, *2*(3), 39-43.

Atzeni, P., Bugiotti, F., Cabibbo, L. & Torlone, R., 2020. Data modeling in the NoSQL world. *Computer Standards & Interfaces*, *67*, p.103149. 1-10.

Atzeni, P., Bugiotti, F. & Rossi, L., 2012. SOS (save our systems) a uniform programming interface for non-relational systems. In *Proceedings of the 15th International Conference on Extending Database Technology,* 582-585.

app-metrics.io, 2021. Metric Types. Available from: https://www.app-metrics.io/getting-started/metric-types [Accessed 25th April 2023].

Baskerville, R., Baiyere, A., Gregor, S., Hevner, A. & Rossi, M., 2018. Design science research contributions: finding a balance between artifact and theory. *Journal of the Association for Information Systems*. *19*(5), 1-16.

Blumhardt N., 2022. Superpower. Available from: https://github.com/datalust/superpower [Accessed 23th January 2023].

Blumhardt N., 2021. Sprache. Available from: https://github.com/sprache/Sprache [Accessed 23th January 2023].

Candel, C.J.F., Ruiz, D.S. & García-Molina, J.J., 2022. A unified metamodel for nosql and relational databases. *Information Systems*, *104*, p.101898, 2-25.

Cox, S., Ahalt, S.C., Balhoff, J., Bizon, C., Fecho, K., Kebede, Y., Morton, K., Tropsha, A., Wang, P. & Xu, H., 2020. Visualization Environment for Federated Knowledge Graphs: Development of an Interactive Biomedical Query Language and Web Application Interface. *JMIR Medical Informatics*, *8*(11), p.e17964, 1-7.

Davoudian, A., Chen, L. & Liu, M., 2018. A survey on NoSQL stores. *ACM Computing Surveys (CSUR)*. *51*(2), 3-36.

Duracik, M., Hrkut, P., Krsak, E. & Toth, S., 2020. Abstract syntax tree based source code antiplagiarism system for large projects set. *IEEE Access*, 8, 175350-175354.

El Maghawry, N. & Dawood, A.R., 2010. Aspect oriented GoF design patterns. In *2010 The 7th International Conference on Informatics and Systems (INFOS)*. IEEE. 1-7.

Endris, K.M., 2020. *Federated Query Processing over Heterogeneous Data Sources in a Semantic Data Lake* (Doctoral dissertation, Universitäts-und Landesbibliothek Bonn), 58-69.

Gadepally, V., Chen, P., Duggan, J., Elmore, A., Haynes, B., Kepner, J., Madden, S., Mattson, T. & Stonebraker, M., 2016. The BigDAWG polystore system and architecture. In *2016 IEEE High Performance Extreme Computing Conference (HPEC),* IEEE, 1-6.

Gahlyan, P. and Singh, S.N., 2018. Analysis of catalogue of GoF software design patterns. In *2018 8th International Conference on Cloud Computing, Data Science & Engineering (Confluence)* . IEEE. 814-818.

Glake, D., Kiehn, F., Schmidt, M., Panse, F. & Ritter, N., 2022. Towards Polyglot Data Stores--Overview and Open Research Questions. *arXiv preprint arXiv:2204.05779,* 1-27.

Gobert, M., 2020. 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, 1-3.

Guo, J., Liu, Q., Lou, J.G., Li, Z., Liu, X., Xie, T. and Liu, T., 2020. Benchmarking meaning representations in neural semantic parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP),* 1520-1528.

Hevner, A., March, S.T., Park, J. & Ram, S., 2004. Design science research in information systems. *MIS quarterly*, *28*(1), 75-105.

Hewasinghage, M., Abelló, A., Varga, J. & Zimányi, E., 2021. Managing polyglot systems metadata with hypergraphs. *Data & Knowledge Engineering*, *134*, p.101896, 1-14.

Kitchenham B, Pretorius R, Budgen D, Brereton OP, Turner M, Niazi M & Linkman S., 2010. Systematic literature reviews in software engineering–a tertiary study. Information and software technology, 52(8), 792-805.

Kitchenham, B., Brereton, O.P., Budgen, D., Turner, M., Bailey, J. & Linkman, S., 2009. Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology*, *51*(1), 7-15.

Khaldi, K., 2017. Quantitative, qualitative or mixed research: Which research paradigm to use?. *Journal of Educational and Social Research*, *7*(2), 15-19.

Khan, Y., Zimmermann, A., Jha, A., Gadepally, V., D'Aquin, M. & Sahay, R., 2019. One size does not fit all: Querying web polystores. *Ieee Access*, 7, 9598-9605.

Khine, P.P. & Wang, Z., 2019. A review of polyglot persistence in the Big Data world. *Information*, *10*(4), 1-19.

Kolev, B., Bondiombouy, C., Levchenko, O., Valduriez, P., Jimenez-Péris, R., Pau, R. & Pereira, J., 2016. Design and implementation of the CloudMdsQL multistore system. In *CLOSER: Cloud Computing and Services Science,* 1, 352-359.

Kolonko, M. & Müllenbach, S., 2020. Polyglot persistence in conceptual modeling for information analysis. In *2020 10th International Conference on Advanced Computer Information Technologies (ACIT)* , IEEE, 590-594.

Košmerl, I., Rabuzin, K. & Šestak, M., 2020. Multi-Model Databases-Introducing Polyglot Persistence in the Big Data World. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO).* IEEE, 1724-1728.

Koutroumanis, N., Kousathanas, N., Doulkeridis, C. & Vlachou, A., 2021. A demonstration of NoDA: unified access to NoSQL stores. *Proceedings of the VLDB Endowment*, *14*(12), 2851-2854.

Mardiana, S., 2020. Modifying Research Onion for Information Systems Research. *Solid State Technology*, *63*(4), 1202-1210.

Merwe, A.V.D., Gerber, A. & Smuts, H., 2019. Guidelines for conducting design science research in information systems. In *Annual Conference of the Southern African Computer Lecturers' Association*. Springer, Cham, 167-178.

Okoli, C., Schabram, K., 2010. A Guide to Conducting a Systematic Literature Review of Information Systems Research, *Sprouts: Working Papers on Information Systems*, 10(26). http://sprouts.aisnet.org/, 10-26.

Olsen, M. & Raunak, M., 2019. Quantitative Measurements of Model Credibility. In *Model Engineering for Simulation*., Academic Press, 163-175.

onlinedatagenerator.com, 2023. Data generation demo page. Available from: https://www.onlinedatagenerator.com/home/demo [Accessed 10th January 2023].

Oussous, A., Benjelloun, F.Z., Lahcen, A.A. & Belfkih, S., 2018. Big Data technologies: A survey. *Journal of King Saud University-Computer and Information Sciences*, *30*(4), 432-436.

Öztürel, İ.A., 2022. Cross-Level Typing The Logical Form For open-domain semantic parsing, 31-38.

Peffers, K., Tuunanen, T., Gengler, C.E., Rossi, M., Hui, W., Virtanen, V. & Bragge, J., 2020. Design Science Research Process: A Model for Producing and Presenting Information Systems Research. *arXiv preprint arXiv:2006.02763*, 84-93.

Pries-Heje, J., Baskerville, R. & Venable, J.R., 2008. Strategies for design science research evaluation. *ECIS 2008 Proceedings, 87*, 1-12

Ramadhan, H., Indikawati, F.I., Kwon, J. & Koo, B., 2020. MusQ: A Multi-store query system for iot data using a datalog-like language. *IEEE Access*, 8, 58032-58050.

Roy-Hubara, N., Shoval, P. & Sturm, A., 2022. Selecting databases for Polyglot Persistence applications. *Data & Knowledge Engineering*, *137*,p.101950, 2-18.

Santana, L.H.Z. & Mello, R.D.S., 2020. Persistence of RDF Data into NoSQL: A Survey and a Unified Reference Architecture. *IEEE Transactions on Knowledge and Data Engineering*, 1-18.

Saunders, M., Lewis, P. & Thornhill, A., 2012. Research Methods for Business Students (Fifth edit). *Essex: Pearson Education Limited,* 106-128.

Tan, R., Chirkova, R., Gadepally, V. & Mattson, T.G., 2017. Enabling query processing across heterogeneous data models: A survey. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 3211-3219.

Vaishnavi, V., Kuechler, W. & Petter, S., 2019. Design science research in information systems, 46-74. Available from: http://desrist.org/desrist/content/design-science-research-in-information-systems.pdf. [Accessed 10th September 2020].

vom Brocke, J., Hevner, A. & Maedche, A., 2020. Introduction to design science research. In *Design Science Research. Cases*. Springer, Cham, 1-17.

Wedyan, F. & Abufakher, S., 2020. Impact of design patterns on software quality: a systematic literature review. *IET Software*, *14*(1), 1-17.

Yang, X., Zhang, X. & Tong, Y., 2022. Simplified abstract syntax tree based semantic features learning for software change prediction. *Journal of Software: Evolution and Process*, *34*(4), p.e2445, 1-9.

Zhang, H., Zhang, C., Hu, R., Liu, X. & Dai, D., 2021. Unified SQL Query Middleware for Heterogeneous Databases. In *Journal of Physics: Conference Series*, IOP Publishing, p.012065, 1873(1), 1-6.

Zhang, M., 2020. A survey of syntactic-semantic parsing based on constituent and dependency structures. *Science China Technological Sciences*, *63*(10), 1898-1920.

Xiao, Y. & Watson, M., 2019. Guidance on conducting a systematic literature review. *Journal of planning education and research*, *39*(1), 93-112.

APPENDICES

APPENDIX A: Redis Schema

<*key*=identity_number>

<*value*=object>

user
identity_number
user_id
student_number
title
other_name
first_name
last_name
birth_date
gender
user_name
psw
ip_address
device
session_id
login_date
logout_date
audit_date
city
country

APPENDIX B: Cassandra Schema

student
id
idno
studentno
title
aka
initials
firstname
lastname
dob
genderid
email
cellno
address
registered
grades

address
streetno
streetname
city
postaladdress
postalcode
province
country

registered
faculty
course
subject
registerdate

grades
subject
mark
symbol

subject
descr
price
period

APPENDIX C: MongoDB Schema

contact
_id: <ObjectId>
emai_address
phone

address
_id: <ObjectId>
number
street
city
code
country

students
_id: <ObjectId>
student_id
student_no
id_number
title
init
name
surname
date_of_birth
gender_identity
contact
address
enroll

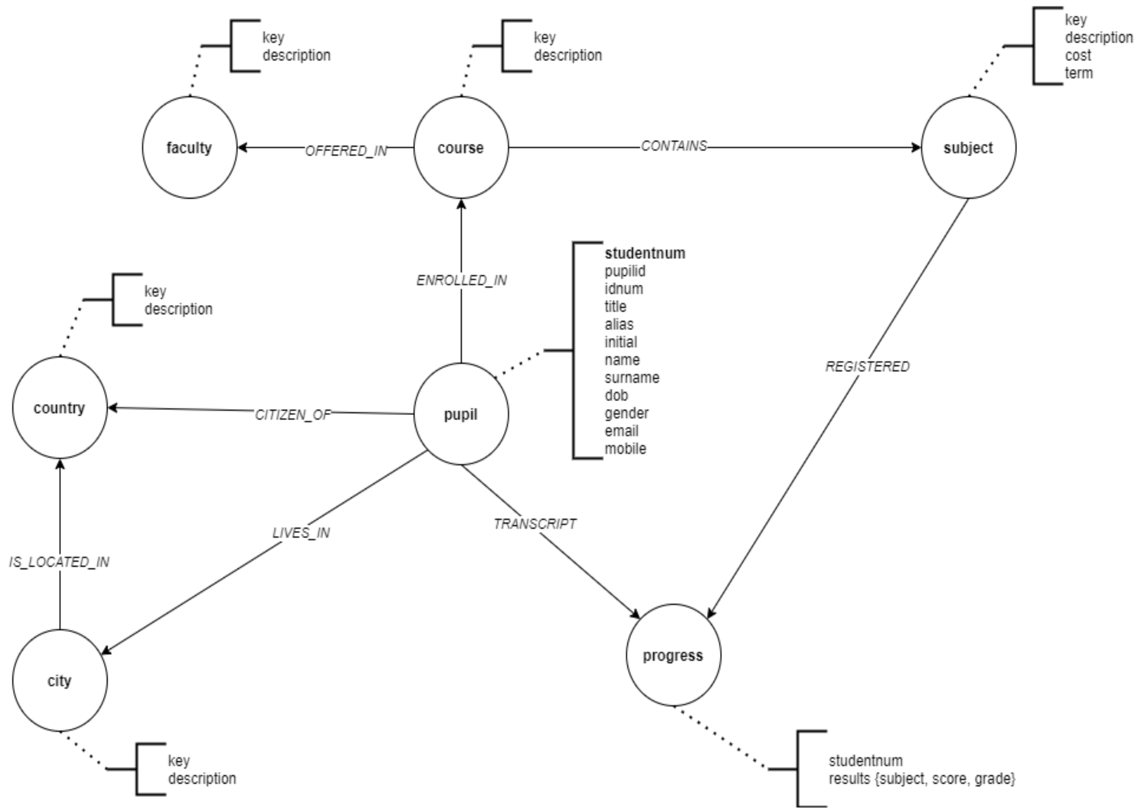
enroll
faculty
course
subject
enrollment_type
enrollment_date

faculty
_id: <ObjectId>
short_code
name

course
_id: <ObjectId>
short_code
name

subject
_id: <ObjectId>
short_code
name
price
duration

APPENDIX D: Neo4j Schema



APPENDIX E: Repository Metamodel

	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 F: Prototype Unified Query - 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 G: Lexer Configuration

	Lexicons	Input Text
Keywords	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

APPENDIX H: 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}*

APPENDIX I: Data Generation - Province

Country	Province\Region
South Africa	"Eastern Cape", "Free State", "Gauteng", "KwaZulu-Natal", "Limpopo", "Mpumalanga", "Northern Cape", "North West", "Western Cape"
Angola	"Bengo", "Benguela", "Bié", "Cabinda", "Cuando Cubango", "Cuanza Norte", "Cuanza Sul", "Cunene", "Huambo", "Huíla", "Luanda", "Lunda Norte", "Lunda Sul", "Malanje", "Moxico", "Namibe", "Uíge", "Zaire"
Nigeria	"Bauchi", "Bida", "Bornu", "Kabba", "Kontagora", "Lower Benue or Nassarawa", "Illorin", "Muri", "Sokoto", "Upper Bema", "Zaria"
Namibia	"Caprivi", "Erongo", "Hardap", "Karas", "Kavango West", "Kavango East", "Khomas", "Kunene", "Ohangwena", "Omaheke", "Omusati", "Oshana", "Oshikoto", "Otjozondjupa"
Botswana	"Central", "Ghanzi", "Kgalagadi", "Kgatleng", "Kweneng", "North East", "North West", "South East", "Southern"
Egypt	"Alexandria Governorate", "Aswan Governorate", "Asyut Governorate", "Beheira Governorate", "Beni Suef Governorate", "Cairo Governorate", "Dakahlia Governorate", "Damietta Governorate", "Faiyum Governorate", "Gharbia Governorate", "Giza Governorate", "Ismailia Governorate", "Kafr El Sheikh Governorate", "Luxor Governorate", "Matruh Governorate", "Minya Governorate", "Monufia Governorate", "New Valley Governorate", "North Sinai Governorate", "Port Said Governorate[5]", "Qalyubia Governorate", "Qena Governorate", "Red Sea Governorate", "Sharqia Governorate", "Sohag Governorate", "South Sinai Governorate", "Suez Governorate"
Tunisia	"Ariana", "Béja", "Ben Arous", "Bizerte", "Gabès", "Gafsa", "Jendouba", "Kairouan", "Kasserine", "Kebili", "Kef", "Mahdia", "Manouba", "Medenine", "Monastir", "Nabeul", "Sfax", "Sidi Bouzid", "Siliana", "Sousse", "Tataouine", "Tozeur", "Tunis", "Zaghouan"

APPENDIX J: Test Cases - Syntax and Sematic Validations

#	Unified Query
87	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno,s.address.street,s.address.postalcode, s.address.postaladdress,s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration,s.register.username, s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } RESTRICT_TO { 1000 } DATA_MODEL { student AS s} TARGET { redis, cassandra, mongodb, neo4j }
88	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, } DATA_MODEL { student AS s} TARGET { redis, cassandra, mongodb, neo4j }
89	FETCH { s.title,s.idnumber,s.newproperty1,s.newproperty2 } DATA_MODEL { student AS s} TARGET { redis, cassandra, mongodb, neo4j }
90	MODIFY { student } FILTER_ON { identifier = '10000' } PROPERTIES { name = 'Tony'} TARGET { redis, cassandra, mongodb, neo4j }

91	PROPERTIES
	{
	identifier='9522896',
	idnumber = '286266761',
	surname = 'Banner',
	name = 'Bruce',
	initial = 'BB',
	gender = 'M',
	title = 'Mr',
	preferredname = 'Hulk'
	}
	ADD { student }
	TARGET { redis, cassandra, mongodb, neo4j }

APPENDIX K: Test Cases - Retrieve complete dataset

#	Unified Query
1	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode,s.address.postaladdress,s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} TARGET { redis }
9	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode,s.address.postaladdress,s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} TARGET { cassandra }
28	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode,s.address.postaladdress,s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} TARGET { mongodb }
45	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode,s.address.postaladdress,s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} RESTRICT_TO { 100 } TARGET { neo4j }
66	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode,s.address.postaladdress,s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} RESTRICT_TO { 1000 } TARGET { redis, cassandra, mongodb, neo4j }

APPENDIX L: Test Cases - Retrieve dataset where a single filter was applied

#	Unified Query
2	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode,s.address.postaladdress,s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} FILTER_ON {s.idnumber = '32502601866'} TARGET { redis }
3	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode,s.address.postaladdress,s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} FILTER_ON {s.gender = 'F'} TARGET { redis }
4	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode,s.address.postaladdress,s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} FILTER_ON { s.idnumber = '11807003413' AND s.gender = 'F'} TARGET { redis }
10	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode,s.address.postaladdress,s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} FILTER_ON {s.identifier = '200'} TARGET { cassandra }
16	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode,s.address.postaladdress,s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s } FILTER_ON { s.gender = 'F'} TARGET { cassandra }

17	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode,s.address.postaladdress,s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s } FILTER_ON { s.name = 'Matthew'} TARGET { cassandra }
54	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street,s.address.postalcode,s.address.postaladdress, s.contact.email, s.contact.mobile} DATA_MODEL { student AS s} FILTER_ON { s.idnumber = '72800706875'} TARGET { neo4j }
67	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street,s.address.postalcode, s.address.postaladdress, s.address.city, s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date } DATA_MODEL { student AS s} FILTER_ON { s.idnumber = '67101803610'} TARGET { redis, cassandra, mongodb, neo4j }
77	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} FILTER_ON { s.transcript.result > 50} TARGET { redis, cassandra, mongodb, neo4j }
78	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} FILTER_ON { s.transcript.result < 50} TARGET { redis, cassandra, mongodb, neo4j }
79	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} FILTER_ON { s.transcript.result >= 20 AND s.transcript.result <= 70} TARGET { redis, cassandra, mongodb, neo4j }

APPENDIX M: Test Cases - Retrieve dataset where a multiples filters were applied

#	Unified Query
11	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode,s.address.postaladdress,s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} FILTER_ON { s.idnumber = '71100307130' AND s.gender = 'M'} TARGET { cassandra }
12	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode,s.address.postaladdress,s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} FILTER_ON { s.idnumber = '71100307130' OR s.gender = 'M'} TARGET { cassandra }
15	FETCH { s.identifier, s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code,s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '71100307130' OR s.gender = 'M'} ORDER_BY { s.identifier} TARGET { cassandra }
29	FETCH { s.identifier, s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code,s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s} FILTER_ON {s.idnumber = '00503100763'} TARGET { mongodb }
30	FETCH { s.identifier, s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code,s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s}

	FILTER_ON { s.idnumber = '51701205088' AND s.gender = 'M'} TARGET { mongodb }
55	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street,s.address.postalcode,s.address.postaladdress, s.contact.email, s.contact.mobile} DATA_MODEL { student AS s} FILTER_ON { s.idnumber = '67101803610' AND s.gender = 'F'} TARGET { neo4j }
56	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street,s.address.postalcode,s.address.postaladdress, s.contact.email, s.contact.mobile} DATA_MODEL { student AS s} FILTER_ON { s.idnumber = '67101803610' OR s.gender = 'M'} RESTRICT_TO { 100 } TARGET { neo4j }

68	FETCH {
	s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date
	}
	DATA_MODEL { student AS s}
	FILTER_ON { s.idnumber = '67101803610' AND s.gender = 'F'}
	TARGET { redis, cassandra, mongodb, neo4j }

69	FETCH {
	s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date
	}
	DATA_MODEL { student AS s}
	FILTER_ON { s.idnumber = '67101803610' OR s.gender = 'M'}
	TARGET { redis, cassandra, mongodb, neo4j }

70	FETCH {
	s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date
	}
	DATA_MODEL { student AS s}
	FILTER_ON { s.idnumber = '67101803610' OR s.gender = 'M'
	AND s.register.studentno = '979883209'}
	TARGET { redis, cassandra, mongodb, neo4j }

80	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name } DATA_MODEL { student AS s} FILTER_ON { s.transcript.symbol = 'A' OR s.transcript.symbol = 'B'} TARGET { redis, cassandra, mongodb, neo4j }
81	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.transcript.symbol } DATA_MODEL { student AS s} FILTER_ON { s.transcript.symbol = 'A' OR s.transcript.symbol = 'B'} TARGET { redis, cassandra, mongodb, neo4j }

APPENDIX N: Test Cases - Apply a limit to the dataset retrieval process

#	Unified Query
13	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city, s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s } RESTRICT_TO { 10 } TARGET { cassandra }
31	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city, s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s } RESTRICT_TO { 10 } TARGET { mongodb }
46	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date } DATA_MODEL { student AS s} RESTRICT_TO { 100 } TARGET { neo4j }
47	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code,s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date } DATA_MODEL { student AS s} RESTRICT_TO { 100 } TARGET { neo4j }
48	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code,s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date } DATA_MODEL { student AS s} RESTRICT_TO { 100 } TARGET { neo4j }
49	FETCH {

	s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.course.code, s.register.course.name, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date } DATA_MODEL { student AS s} RESTRICT_TO { 100 } TARGET { neo4j }
50	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city, s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile, s.register.studentno, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date } DATA_MODEL { student AS s} RESTRICT_TO { 100 } TARGET { neo4j }
51	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile} DATA_MODEL { student AS s} RESTRICT_TO { 100 } TARGET { neo4j }
52	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street,s.address.postalcode,s.address.postaladdress,s.address.city, s.contact.email, s.contact.mobile} DATA_MODEL { student AS s} RESTRICT_TO { 100 } TARGET { neo4j }
53	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.contact.email, s.contact.mobile} DATA_MODEL { student AS s} RESTRICT_TO { 100 } TARGET { neo4j }
54	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.contact.email, s.contact.mobile} DATA_MODEL { student AS s} FILTER_ON { s.idnumber = '72800706875'} TARGET { neo4j }

APPENDIX O: Test Cases - Apply sorting to the dataset retrieval process

#	Unified Query
14	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city, s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '71100307130' OR s.gender = 'M'} ORDER_BY { s.name} TARGET { cassandra }
32	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s } ORDER_BY { s.name} TARGET { mongodb }
33	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '58308108421'} ORDER_BY { s.name} TARGET { mongodb }
34	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '78702504377' AND s.register.studentno = '779529903'} ORDER_BY { s.name} TARGET { mongodb }
35	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '24106105288' OR s.idnumber = '88404705416'} ORDER_BY { s.name} TARGET { mongodb }
36	FETCH {

	s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress, s.address.city,s.address.country.code, s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date,s.transcript.subject, s.transcript.result, s.transcript.symbol } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '24106105288' OR s.idnumber = '88404705416' AND s.gender = 'F' } ORDER_BY { s.name} TARGET { mongodb }
57	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street,s.address.postalcode,s.address.postaladdress, s.contact.email, s.contact.mobile} DATA_MODEL { student AS s} FILTER_ON { s.gender = 'M'} RESTRICT_TO { 100 } ORDER_BY { s.surname, s.name} TARGET { neo4j }
71	FETCH { s.title,s.idnumber,s.preferredname,s.initial,s.name, s.surname, s.dateofbirth, s.gender, s.address.streetno, s.address.street, s.address.postalcode, s.address.postaladdress,s.address.city,s.address.country.code,s.address.country.name, s.contact.email, s.contact.mobile,s.register.studentno, s.register.faculty.code, s.register.faculty.name, s.register.course.code, s.register.course.name, s.register.subject.code, s.register.subject.name, s.register.subject.cost, s.register.subject.duration, s.register.username,s.register.password, s.register.type, s.register.ipaddress, s.register.date } DATA_MODEL { student AS s} FILTER_ON { s.idnumber = '67101803610' OR s.gender = 'M' AND s.register.studentno = '979883209'} ORDER_BY { s.surname, s.idnumber} TARGET { redis, cassandra, mongodb, neo4j }

APPENDIX P: Test Cases - Aggregation on a datasets

#	Unified Query
19	FETCH{ s.idnumber, s.initial,s.name, s.surname,s.transcript.subject,NAVG(s.transcript.result)} DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '77607500615'} TARGET { cassandra }
20	FETCH { s.idnumber, s.initial, s.name, s.surname, s.transcript.subject, NCOUNT(s.transcript.result) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '77607500615'} TARGET { cassandra }
21	FETCH { s.idnumber, s.initial, s.name, s.surname, s.transcript.subject, NMIN(s.transcript.result) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '77607500615'} TARGET { cassandra }
22	FETCH { s.idnumber, s.initial, s.name, s.surname, s.transcript.subject, NMAX(s.transcript.result) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '77607500615'} TARGET { cassandra }
37	FETCH {s.idnumber, s.initial, s.name, s.surname, NSUM(s.register.subject.cost) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '58602700606'} TARGET { mongodb }
38	FETCH {s.idnumber, s.initial,s.name, s.surname,NAVG(s.register.subject.cost)} DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '58602700606'} TARGET { mongodb }
39	FETCH{s.idnumber,s.initial,s.name,s.surname,COUNT(s.register.subject.cost) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '58602700606'} TARGET { mongodb }
40	FETCH {s.idnumber, s.initial,s.name, s.surname, NMIN(s.register.subject.cost)} DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '58602700606'} TARGET { mongodb }
41	FETCH {s.idnumber, s.initial, s.name, s.surname, MAX(s.register.subject.cost) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '58602700606'} TARGET { mongodb }
58	FETCH {s.idnumber, s.initial, s.name, s.surname, NSUM(s.transcript.result) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '88404705416'} TARGET { neo4j }
59	FETCH {s.idnumber, s.initial, s.name, s.surname, NAVG(s.transcript.result) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '88404705416'} TARGET { neo4j }
60	FETCH {s.idnumber, s.initial, s.name, s.surname, NCOUNT(s.transcript.result) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '88404705416'} TARGET { neo4j }
61	FETCH {s.idnumber, s.initial, s.name, s.surname, NMIN(s.transcript.result) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '88404705416'} TARGET { neo4j }
62	FETCH {s.idnumber, s.initial, s.name, s.surname, NMAX(s.transcript.result) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '88404705416'} TARGET { neo4j }
72	FETCH {s.idnumber, s.initial, s.name, s.surname, NSUM(s.transcript.result) }

	DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '21708702176'} TARGET { redis, cassandra, mongodb, neo4j }
73	FETCH {s.idnumber, s.initial, s.name, s.surname, NAVG(s.transcript.result) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '21708702176'} TARGET { redis, cassandra, mongodb, neo4j }
74	FETCH {s.idnumber, s.initial, s.name, s.surname, NCOUNT(s.transcript.result) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '21708702176'} TARGET { redis, cassandra, mongodb, neo4j }
75	FETCH {s.idnumber, s.initial, s.name, s.surname, NMIN(s.transcript.result) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '21708702176'} TARGET { redis, cassandra, mongodb, neo4j }
76	FETCH {s.idnumber, s.initial, s.name, s.surname, NMAX(s.transcript.result) } DATA_MODEL { student AS s } FILTER_ON { s.idnumber = '21708702176'} TARGET { redis, cassandra, mongodb, neo4j }

APPENDIX Q: Test Cases - Update existing dataset

#	Unified Query
5	MODIFY { student } PROPERTIES { gender = 'M'} FILTER_ON { idnumber = '34502402028||'} TARGET { redis }
6	MODIFY { student } PROPERTIES { register.username = 'newuser', password = 'newpassword'} FILTER_ON { idnumber = '47803702771'} TARGET { redis }
23	MODIFY { student } PROPERTIES { name = 'Test 1', surname = 'Test 2', initial = 'TT'} FILTER_ON { identifier = '5' } TARGET { cassandra }
24	MODIFY { student } PROPERTIES { name = 'Micheal', surname = 'Corleone', initial = 'M'} FILTER_ON { idnumber = '65500804135'} TARGET { cassandra }
25	MODIFY { student } PROPERTIES { name = 'John', surname = 'Doe', initial = 'JD'} FILTER_ON { name = 'Micheal'} TARGET { cassandra }
42	MODIFY { student } PROPERTIES { name = 'Jane', surname = 'Doe', initial = 'JD'} FILTER_ON { idnumber = '83604407222'} TARGET { mongodb }
43	MODIFY { student } PROPERTIES { name = 'Jane-Anne', surname = 'Jenkins', initial = 'JA'} FILTER_ON { idnumber = '57508002711' AND register.studentno = '391050029'} TARGET { mongodb }
63	MODIFY { student } PROPERTIES { name = 'Jane', surname = 'Doe', initial = 'JD'} FILTER_ON { idnumber = '83604407222'} TARGET { neo4j }
64	MODIFY { student } PROPERTIES { name = 'Jane-Anne', surname = 'Jenkins', initial = 'JA'} FILTER_ON { idnumber = '57508002711' AND register.studentno = '391050029'} TARGET { neo4j }
82	MODIFY { student } PROPERTIES { name = 'Mary', surname = 'Poppins', initial = 'MP'} FILTER_ON { idnumber = '85208201670'} TARGET { redis, cassandra, mongodb, neo4j }
83	MODIFY { student } PROPERTIES { name = 'Clark', surname = 'Kent', initial = 'Mel', title = 'MR', preferredname = 'Superman'} FILTER_ON { idnumber = '75602501070'} TARGET { redis, cassandra, mongodb, neo4j }

84	MODIFY { student }
	PROPERTIES {
	name = 'Clark', surname = 'Kent',
	initial = 'Mel',
	gender = 'M',
	title = 'MR',
	preferredname = 'Superman'
	}
	FILTER_ON { identifier = '10000' }
	TARGET { redis, cassandra, mongodb, neo4j }

85	MODIFY { student } PROPERTIES { dateofbirth ='1970/10/13'} FILTER_ON { identifier = '10000' } TARGET { redis, cassandra, mongodb, neo4j }

APPENDIX R: Test Cases - Data inserts

#	Unified Query
7	ADD { student } PROPERTIES { idnumber = '564379484', name = 'Chuck T', surname = 'Tester'} TARGET { redis }
8	ADD { student } PROPERTIES { name = 'Chuck T', surname = 'Tester'} TARGET { redis }
26	ADD { student } PROPERTIES { identifier = '323323995', idnumber = '876765564431', title = 'Miss', name = 'Lauren', surname = 'Cole', register.studentno = '7149222' } TARGET { cassandra }
44	ADD { student } PROPERTIES { idnumber = '6062390', title = 'Miss', name = 'Lauren', surname = 'Cole', register.studentno = '53012' } TARGET { mongodb }
	ADD { student } PROPERTIES { idnumber = '8078891', title = 'Miss', name = 'Lauren', surname = 'Cole'} TARGET { neo4j }
86	ADD { student } PROPERTIES { identifier='2913511', idnumber = '980180616', surname = 'Banner', name = 'Bruce', initial = 'BB', gender = 'M', title = 'Mr', dateofbirth ='1970/10/13' preferredname = 'Hulk'} TARGET { redis, cassandra, mongodb, neo4j }

APPENDIX S: Apdex Nonparametric Correlations

Correlation Parameters

	Max	Mean	Median	Min	Percentile	stdDev	Sum	Fifteen Minute Rate	Five Minute Rate	Mean Rate	One Minute Rate
	1	.894**	.877**	.868**	1.000**	-.159*	.946**	-0.021	-0.021	-0.008	-0.021
	.	<.001	<.001	<.001	.	0.011	<.001	0.738	0.738	0.894	0.738
	256	256	256	256	256	256	256	256	256	256	256
	.894**	1	.992**	.994**	.894**	-.569**	.758**	-.288**	-.288**	-.266**	-.288**
	<.001	.	<.001	<.001	<.001	<.001	<.001	<.001	<.001	<.001	<.001
	256	256	256	256	256	256	256	256	256	256	256
	.877**	.992**	1	.992**	.877**	-.584**	.747**	-.295**	-.295**	-.273**	-.295**
	<.001	<.001	.	<.001	<.001	<.001	<.001	<.001	<.001	<.001	<.001
	256	256	256	256	256	256	256	256	256	256	256
	.868**	.994**	.992**	1	.868**	-.611**	.732**	-.312**	-.312**	-.289**	-.312**
	<.001	<.001	<.001	.	<.001	<.001	<.001	<.001	<.001	<.001	<.001
	256	256	256	256	256	256	256	256	256	256	256
	1.000**	.894**	.877**	.868**	1	-.159*	.946**	-0.021	-0.021	-0.008	-0.021
	.	<.001	<.001	<.001	.	0.011	<.001	0.738	0.738	0.894	0.738
	256	256	256	256	256	256	256	256	256	256	256
	-.159*	-.569**	-.584**	-.611**	-.159*	1	0.083	.621**	.621**	.592**	.621**
	0.011	<.001	<.001	<.001	0.011	.	0.186	<.001	<.001	<.001	<.001
	256	256	256	256	256	256	256	256	256	256	256
	.946**	.758**	.747**	.732**	.946**	0.083	1	.157*	.157*	.163**	.157*
	<.001	<.001	<.001	<.001	<.001	0.186	.	0.012	0.012	0.009	0.012
	256	256	256	256	256	256	256	256	256	256	256
	-0.021	-.288**	-.295**	-.312**	-0.021	.621**	.157*	1	1.000**	.987**	1.000**
	0.738	<.001	<.001	<.001	0.738	<.001	0.012	.	.	<.001	.

	256	256	256	256	256	256	256	256	256	256	256
	-0.021	-.288**	-.295**	-.312**	-0.021	.621**	.157*	1.000**	1	.987**	1.000**
	0.738	<.001	<.001	<.001	0.738	<.001	0.012	.	.	<.001	.
	256	256	256	256	256	256	256	256	256	256	256
	-0.008	-.266**	-.273**	-.289**	-0.008	.592**	.163**	.987**	.987**	1	.987**
	0.894	<.001	<.001	<.001	0.894	<.001	0.009	<.001	<.001	.	<.001
	256	256	256	256	256	256	256	256	256	256	256
	-0.021	-.288**	-.295**	-.312**	-0.021	.621**	.157*	1.000**	1.000**	.987**	1
	0.738	<.001	<.001	<.001	0.738	<.001	0.012	.	.	<.001	.
	256	256	256	256	256	256	256	256	256	256	256

APPENDIX T: Source Code

#	
Project Name	Unified Query Prototype
Description	This project was created as part of the researcher MICT degree at CPUT.
Location	https://github.com/hadwinv/CPUT.Polyglot.NoSql.git