

Development of a Dynamic Design Framework
for Relational Database Performance Optimisation

Mr. Derek Andrew Colley

Student identifier: 16025973

Principal Supervisor: Dr. Md Asaduzzaman
Second Supervisor: Mr. Euan Wilson

A thesis submitted in partial fulfilment of the requirements of
Staffordshire University for the degree of Doctor of Philosophy

Submitted February 2021

Amended with corrections, August 2021

Word count: 79,350, excluding Appendices.

Abstract

Relational Database Management Systems (RDBMSs) are advanced software packages responsible for providing storage and access to relational databases; data stores in which data is arranged in schemas, which are interlinked tables, each table constituted of columns and rows, and each intersection containing a data point.

This project considers the impact that the ever-increasing demand in data volume, velocity and variety, combined with changes in query methodology and uptake of object-relational mapping frameworks driven by modern object-oriented application programming practices, have had upon the effectiveness of the relational database query optimiser; in particular, this research examines the emergence of object-relational impedance mismatch and the corresponding effect on query processing efficiency within the database engine.

Firstly, this research reconsiders the query parsing and caching mechanisms within current RDBMSs and notes their deficiencies in query plan re-use. An alternative mechanism for query representation is presented, representing queries as multidimensional structures which are computable, comparable, and reducible to hashes. It is shown how this representation can be used to improve plan re-use and increase the efficiency of the query optimiser.

Secondly, new multidimensional representations in real-time are demonstrated using weighted k -means clustering with self-adjusting weights and k to predict superior sub-schema selection, including application of queries to an alternative sub-schema of data, reducing resource consumption and improving query execution times. This is validated against a real data set and performance is tested at scale. It was found that use of KNN provided the relational database query optimiser with an increasing degree of accuracy and reliability in query classification, with an improvement in query execution time demonstrated at scale, against lifelike database queries, ranging from 6.2% to 20.6%.

Finally, a novel method of dynamic schema redefinition is presented. This process defines, creates and destroys sub-schemas, maps queries to their sub-schema variants, and keeps track of performance metrics, self-adjusting the current library of alternative schema representations available. This is defined theoretically against the backdrop of the relational algebra and ZFC axiomatic set theory.

Acknowledgements

I would like to acknowledge and thank my supervision team:

Dr. Md Asaduzzaman, Euan Wilson, and Dr. Clare Stanier
for their superb advice, direction, and support throughout this project.

List of Publications

- Colley, D. and Asaduzzaman, M. (2021). ‘A Novel Method for Calculating Query Hashes for Improved Query Grouping in Relational Database Management Systems’. *Springer Nature - Book Series: Transactions on Computational Science & Computational Intelligence (and 21st International Conference on Information and Knowledge Engineering)*. In press. Available at: <https://tinyurl.com/2ac6uxnz> (Accessed 02 August 2021).
- Colley, D. and Asaduzzaman, M. (2020). ‘Derivation of Dynamic Schema Definitions from Query Patterns in Relational Databases for Improved Query Execution Efficiency’. Preprint. Available at: <https://tinyurl.com/y53eeapt> (Accessed 18 January 2021).
- Colley, D., Stanier, C., and Asaduzzaman, M. (2020). ‘Investigating the Effects of Object-Relational Impedance Mismatch on the Efficiency of Object-Relational Mapping Frameworks’. *Journal of Database Management*, 31(4). Available at: <https://www.igi-global.com/article/investigating-the-effects-of-object-relational-impedance-mismatch-on-the-efficiency-of-object-relational-mapping-frameworks/266402> (Accessed 18 January 2021).
- Colley, D., Stanier, C. and Asaduzzaman, M. (2018). ‘The Impact of Object-Relational Mapping Frameworks on Relational Query Performance’. *International Conference on Computing, Electronics & Communications Engineering (ICCECE '18)*. Available at: <https://ieeexplore.ieee.org/document/8659222> (Accessed 18 January 2021).
- Colley, D. and Stanier, C. (2017). ‘Identifying New Directions in Database Performance Tuning’. *Procedia Computer Science*, vol. 121, pp.260-265. Available at: <https://www.sciencedirect.com/science/article/pii/S1877050917322275> (Accessed 18 January 2021).

Table of Contents

| | |
|---|----|
| Abstract | 1 |
| Acknowledgements | 2 |
| List of Publications | 3 |
| Table of Contents | 4 |
| List of Tables, Figures, Algorithms and Code Listings | 7 |
| Chapter 1: Research Introduction | 8 |
| 1.1 Introduction | 8 |
| 1.2 Research Motivation | 8 |
| 1.4 Research Questions, Aims and Objectives | 10 |
| 1.4 Research Approach | 12 |
| 1.5 Ethical Issues | 25 |
| 1.6 Thesis Structure | 26 |
| 1.7 Novel Contributions to Knowledge | 28 |
| 1.8 Chapter Summary | 29 |
| Chapter 2: Background | 30 |
| 2.1 Introduction | 30 |
| 2.2 Relational Database Management Systems | 30 |
| 2.3 Query Representation and Comparison | 33 |
| 2.4 Query Representation and Execution | 37 |
| 2.5 Query Optimisation | 43 |
| 2.6 Chapter Summary | 48 |
| Chapter 3: Literature Review | 50 |
| 3.1 Introduction | 50 |
| 3.2 Literature Review Methodology | 50 |
| 3.3 Database Performance Tuning | 50 |
| 3.4 Query Tuning and Frameworks | 54 |
| 3.5 Existing Query Parsing Techniques | 56 |
| 3.6 Object-Relational Mapping Technologies | 4 |
| 3.8 Conclusions | 11 |
| 3.9 Chapter Summary | 12 |
| Chapter 4 - Problem Investigation | 13 |
| 4.1 Introduction | 13 |
| 4.2 Domain Expert Investigation - Survey | 14 |
| 4.3 Domain Expert Investigation - Interviews | 19 |

| | | |
|--|--|-----|
| 4.4 | Conclusions from Domain Expert Investigations | 24 |
| 4.5 | Experimental Investigations | 26 |
| 4.6 | Chapter Summary | 43 |
| Chapter 5 - Solution Design | | 45 |
| 5.1 | Introduction | 45 |
| 5.2 | Context | 45 |
| 5.3 | Solution Overview | 46 |
| 5.4 | Principal PETAS components | 47 |
| 5.5 | Queries as Graphs – the Query Parser and Similarity Scorer | 52 |
| 5.6 | The Schema Classifier and Query Mapper | 66 |
| 5.7 | Dynamic Schema Redefinition | 69 |
| 5.8 | Chapter Summary | 77 |
| Chapter 6 – Testing: Query Representation | | 78 |
| 6.1 | Introduction | 78 |
| 6.2 | Design | 79 |
| 6.2 | Algorithmic Implementation | 84 |
| 6.3 | Practical Implementation | 89 |
| 6.4 | Experimental Design | 94 |
| 6.5 | Testing and Results | 97 |
| 6.6 | Conclusions | 98 |
| 6.7 | Chapter Summary | 99 |
| Chapter 7 – Testing: Similarity Scoring and Schema Selection | | 100 |
| 7.1 | Introduction | 100 |
| 7.2 | Algorithmic Implementation | 104 |
| 7.3 | Practical Implementation | 110 |
| 7.4 | Experimental Design | 111 |
| 7.5 | Testing and Results | 118 |
| 7.6 | Conclusions | 2 |
| 7.7 | Chapter Summary | 3 |
| Chapter 8 – Testing: Dynamic Schema Redefinition | | 4 |
| 8.1 | Introduction | 4 |
| 8.2 | Algorithmic Implementation | 4 |
| 8.3 | Practical Implementation | 7 |
| 8.4 | Experimental Design | 9 |
| 8.5 | Testing and Results | 11 |
| 8.6 | Conclusions | 19 |
| 8.7 | Chapter Summary | 21 |
| Chapter 9: Conclusions and Future Work | | 23 |

| | | |
|-----|---|-----|
| 9.1 | Introduction | 23 |
| 9.2 | Problem Investigation | 23 |
| 9.3 | Query Representation | 26 |
| 9.4 | Similarity Scoring and Schema Selection | 28 |
| 9.5 | Dynamic Schema Redefinition | 30 |
| 9.7 | Assessing the Research Questions, Aims and Objectives | 33 |
| 9.8 | Future Research Directions | 39 |
| 9.9 | Chapter Summary | 40 |
| | References | 41 |
| | Chapter 1 | 41 |
| | Chapter 2 | 43 |
| | Chapter 3 | 47 |
| | Chapter 4 | 53 |
| | Chapter 5 | 54 |
| | Chapter 6 | 56 |
| | Chapter 7 | 56 |
| | Chapter 8 | 56 |
| | Chapter 9 | 57 |
| | Appendix A – Practitioner Survey Structure | 58 |
| | Appendix B: Strong sentiment groupings from interview analysis | 80 |
| | Appendix C: Query objectives and code listings from the initial investigation | 85 |
| | Appendix D: Similarity scoring and schema selection – code listings | 89 |
| | Appendix E: Dynamic schemas – algorithms and code | 102 |
| E.1 | Implementation of the query parser component | 102 |
| E.2 | Temporary table creation | 104 |
| E.3 | Implementation of the analyse MVs/use metadata component | 106 |
| E.4 | Implementation of the create and destroy MVs component | 115 |
| E.5 | Implementation of the Query Generator and Caller (for Testing Purposes) | 130 |

List of Tables, Figures, Algorithms and Code Listings

| | | | |
|---|-----|---|-----|
| Chapter 1 | | Chapter 6 (cont'd) | |
| Fig. 1.1: The 'Research Onion' | 15 | Alg.6.10: Funct'n variant of query parser | 148 |
| Fig. 1.2: Three-Phase Research Plan | 18 | Alg. 6.11: Converting edge list to a cube | 149 |
| Tab. 1.3: A sample of topics | 19 | Cd. Lstg. 6.12: Algorithm 1 in Python | 149 |
| Fig. 1.4: Grounded-theory approach | 20 | Cd. Lstg. 6.13: Algorithm 2 in Python | 149 |
| Fig. 1.5: The original spiral methodology | 24 | Cd. Lstg. 6.14: Algorithm 3 in Python | 150 |
| | | Cd. Lstg. 6.15: Algorithm 4 in Python | 151 |
| Chapter 2 | | Fig. 6.16: Screenshot from edge builder | 152 |
| Fig. 2.1: Illustration of a theta-join | 35 | Cd. Lstg. 6.17: Conversion to multi. array | 152 |
| Fig. 2.2: The query execution cycle | 38 | Cd. Lstg. 6.18: Fun'd cube build code | 153 |
| Fig. 2.3: Two execution plans compared | 40 | Cd. Lstg. 6.19: Randm'd query generator | 154 |
| Fig. 2.4: Illust. of a normalised database query | 45 | Fig. 6.20: Randomly generated queries | 156 |
| Fig. 2.5: Illust. of a de-normalised DB query | 45 | Tab. 6.21: Funct'l transformat'n testing | 157 |
| Tab. 2.6: Relative costs comp'd between queries | 46 | Fig. 6.22: Duration statistics | 157 |
| | | Chapter 7 | |
| Chapter 3 | | Tab. 7.1 and 7.2: Edge lists for Q1 and Q2 | 161 |
| Fig. 3.1: Parse tree illustration | 60 | Fig. 7.3: Adjacency cube for query 1 | 161 |
| Fig. 3.2: Query tokenisation example | 61 | Fig. 7.4: Adjacency cube for query 2 | 161 |
| | | Fig. 7.5: Resulting adjacency cube C3 | 162 |
| Chapter 4 | | Fig. 7.6: Similar'y scor'g & query mapper | 163 |
| Tab. 4.1: Final codification of the survey results | 76 | Alg. 7.7: The similarity scoring algorithm | 164 |
| Fig. 4.2: Survey outcomes as a thematic map | 77 | Tab. 7.8: Query cache table design | 165 |
| Tab. 4.3: Mapping i'view questions to themes | 80 | Fig. 7.9: The KNN classifier concept | 166 |
| Fig. 4.4: Frequency breakdown of survey topics | 81 | Alg. 7.10: Looping ...the query cache | 166 |
| Fig. 4.5: Execution plan for the ORM query | 88 | Alg. 7.11: Finding similar queries | 167 |
| Fig. 4.6: Execution plan for the non-ORM query | 89 | Tab. 7.12: Query stack table design | 168 |
| Fig. 4.7: Execution plan ... ORM search query | 91 | Alg. 7.13: Asynch's weight adjustment | 168 |
| Fig. 4.8: Execution plan for the non-ORM query | 92 | Alg. 7.14: Adjusting the value of K | 169 |
| Tab. 4.9: Performance statistics for Contoso | 93 | Tab. 7.15: Chicago Public Safety data set | 172 |
| Tab. 4.10: Measures / compare effic'y of queries | 96 | Fig. 7.16: Chicago data split/sub-schemas | 174 |
| Tab. 4.11: Results from query perf. testing | 98 | Fig. 7.17: Chicago data table cardinalities | 175 |
| Fig. 4.12: Total Score by Evaluation Criterion | 100 | Fig. 7.18: Output-random SQL generator | 177 |
| Fig. 4.13: Total Score by Query Objective | 101 | Tab. 7.19: Results from scoring testing | 179 |
| Fig. 4.14: Correlation bet'n complexity and time | 102 | Tab. 7.20: Failed query mappings | 180 |
| Tab. 4.15: <i>p</i> -values / t-testing execution times | 102 | Tab. 7.21: Test descriptions | 183 |
| | | Figs. 7.22(a), 7.22(b): Processing metrics | 184 |
| Chapter 5 | | Fig. 7.23: Cost savings / query/ schema | 185 |
| Fig. 5.1: The query execution lifecycle | 106 | Figs. 7.24(a), 7.24(b): Distr'n, correlation | 187 |
| Fig. 5.2: Overview of PETAS | 108 | Chapter 8 | |
| Fig. 5.3: The alt. query representation process | 109 | Fig. 8.1: Overview of dynamic schemas | 190 |
| Fig. 5.4: Example SQL query / ERD diagram | 114 | Fig. 8.2: DFD for dynamic schema process | 190 |
| Fig. 5.5: Distinct query compn't list (key-value) | 115 | Fig. 8.3: ERD for dynamic schema tables | 192 |
| Fig. 5.6(a,b): Adjacency matrix and graph | 116 | Tab. 8.4: Query parser summary metrics | 199 |
| Fig. 5.7: Query tokenisation flowchart | 117 | Tab. 8.5: Analyse MVs s'ry metrics I | 200 |
| Fig. 5.8: Directed graph representations | 121 | Tab. 8.6: Analyse MVs s'ry metrics II | 200 |
| Fig. 5.9: Calculating Hamming distances | 122 | Tab. 8.7: All phases – summary metrics | 200 |
| Fig. 5.10: Calculating the adjacency cubes | 123 | Fig. 8.8: Cost deltas / all queries, all runs | 201 |
| Fig. 5.11(a-c): K in one/two/three dimensions | 125 | Tab. 8.9: Metrics for ... cost delta measure | 201 |
| Tab. 5.12: Calculating traversal cost | 133 | Fig. 8.10: Cost deltas - new q'ry cost lower | 202 |
| Tab. 5.13: Relative index seek efficiency | 134 | Tab. 8.11: Metrics ... cost delta measure | 202 |
| | | Tab. 8.12: Metrics for proc. queries | 203 |
| Chapter 6 | | Tab. 8.13: Metrics for filtered queries | 203 |
| Fig. 6.1: Overview of PETAS – matrix parser | 138 | Tab. 8.14: Storage required by new MVs | 204 |
| Fig. 6.2: Visualising a query in 3 dimensions | 139 | Chapter 9 | |
| Tab. 6.3: Example node relationship list | 140 | No figures. | |
| Tab. 6.4: 2-dimensional adjacency matrix | 141 | | |
| Fig. 6.5: Attribute type on the Z-axis | 141 | | |
| Fig. 6.6: 3D adjacency cube in 2 dimensions | 143 | | |
| Alg. 6.7: Extracting projection elements | 144 | | |
| Alg. 6.8: Extracting membership elements | 145 | | |
| Alg. 6.9: Extracting predicate elements | 146 | | |
| | | + <i>Appendices A-E (various)</i> | |

Chapter 1: Research Introduction

1.1 Introduction

Relational Database Management Systems (henceforth RDBMSs) manage storage and access to data using the relational model, accessible through Structured Query Language (SQL) both directly by users and automatically via application calls. Such systems underpin a large majority of the everyday IT systems used across the world [1], from physical barrier controls to e-commerce websites, stock market systems and social media. At over half a century old [2], the relational model has stood the test of time and has spawned a diverse range of powerful toolsets espoused by keystone software suppliers; Microsoft, Oracle and IBM all have contemporary flagship RDBMS products [3, 4, 5] that have a rich history, with more recent inroads made with ‘Database-as-a-Service’ (DaaS) products from Amazon and developments in both native and DaaS RDBMS platforms from the open-source community. However, the relational model does not cater for every usage scenario, and several classes of performance issues are endemic to the model, as evidenced throughout the academic literature [6, 7] and through observations and improvements suggested by the technical practitioner community [8, 9].

Chapters 2 and 3 explore the literature and show primary research conducted to replicate some well-known database query performance anti-patterns, particularly those investigated by Ireland et al. [6] and Karwin [10]. Many of these issues remain current and merit further research, particularly in the efficiency of query parsing, caching and binding. The principal research contributions explore these issues, replicating and exploring poor performance characteristics in an experimental setting, and present a novel, multi-faceted solution framework that represents advances in both the information theory underpinning the representation of relational queries and the pragmatic delivery of a new methodology for query handling within the RDBMS query engine. This research aims to both showcase the theoretical development of the novel ideas that underpin this methodology and demonstrate how the framework can be implemented within an RDBMS.

1.2 Research Motivation

The role of information has undergone a radical transformation since the inception of the database model in 1970, both in the context of technological development of information systems and as part of wider cultural changes in the way that information is produced, stored, and consumed. The development of web applications, the advent of social media and the increase in development and proliferation of online appliances (known colloquially as the ‘Internet of Things’[11]) in a variety of

contexts, such as enabling so-called ‘smart cities’ [12] or powering developments in healthcare [13, 14], together with sustained improvements in computational capability, have resulted in what has been described variously as a ‘data deluge’[15]; the ‘information revolution’ [16] and the ‘global information age’ [17]. However, as data is generated, it must be stored, and must then be kept *confidential*; the *integrity* of the data must be preserved; and it must be *available*, ready to be accessed. This CIA triad forms the core tasks of the RDBMS. Ensuring these challenges are met is increasingly difficult in a rapidly changing world where the use and generation of data is continually growing.

Traditionally, the RDBMS has been used to store such data and provide this ‘CIA guarantee’, on the premise that the data is structurally repeatable, that it conforms to a formal data schema. This remains the case for a great deal of enterprise data; of the top 10 databases in the world, ranked by popularity, 7 are relational platforms [1]. It is notable that one of the world’s most popular social networks is constructed on a relational database product and in 2011 was handling over 60 million SQL queries per second [8]. While there are valid use cases for non-relational database systems, such as the management of unstructured or semi-structured data, it is arguable that the widespread implementation of RDBMSs in diverse contexts means relational database performance remains a current concern and NoSQL solutions, while suitable for some purposes, cannot provide a superior fit between the underpinnings of set-theoretic relational algebra and implementations of the same than the relational model paradigm.

Database schemas are noted for their invariability [18]. Having been conceived and the standards developed from 1970 onwards [2][19][20], it is symptomatic of the static nature of computer programming at the time that the relational model was designed to integrate with fixed application models. However, technical and cultural patterns in modern application development are designed with adaptability in mind, a major milestone of which was the publication of the ‘Agile Manifesto’ [21] in 2001 and led to a shift away from application development methodologies that had ‘Big Design Up Front’ (BDUF) principles embedded. This included to some extent the relational model, built to interface with these kinds of systems, and solutions optimised for semi-structured data built on the BASE rather than the ACID principles became popular [22, 23]. As such, both technical and cultural splits developed between the new object-oriented, adaptable application development approaches and the static relational model. One artefact of this technical split is known as the object-relational impedance mismatch problem [6, 24], necessitating the use of object-relational mapping (ORM) tools to reconcile class objects to sets, while the cultural split has resulted in various speculations across social, news and academic media [7, 25] about the role of relational databases in future applications and helped fuel the rise of alternative database platforms that

support unstructured and semi-structured data.

With this context described, the motivation for this research is to help solve, or at least mitigate, some of these mismatch and subsequent performance issues by introducing *dynamism* – defined here as the ability for a RDBMS to respond favourably to a constantly-changing environment - into the relational model, as agility was introduced into software development practices. To do so, this research focuses on novel theoretical developments and demonstrable practical methods which enable the database to respond to ever-changing input queries to provide a superior data retrieval service which is better placed to serve the management of ever-increasing volumes of data.

1.4 Research Questions, Aims and Objectives

This section lists the broad research questions and the central aims of this research project; and details several objectives that support the aims and look to answer the research questions. These are revisited in Chapter 11, where the findings are compared against the stated questions, aims and objectives, and the success (or otherwise) of the project is evaluated.

1.4.1 *Research questions*

- a. As the demands of data processing have evolved from closed systems with known data structures driven by fixed schemas to open, unstructured systems driven by the applications, what disadvantages can be identified with the current object-relational database model given this evolution, and how can these be overcome?
- b. Can a new theory for query representation be developed as an alternative to representing queries as semantic objects? Is there an accompanying viable practical approach to implementing this new theory to overcome the disadvantages of storing and caching queries as non-comparable semantic objects, and can this be used to improve the parsing and pre-optimisation stages of the query optimiser?
- c. Can other approaches from alternative computational disciplines, such as machine learning, be applied to extend the current object-relational database storage and management methodologies, creating a responsive model that learns from system inputs to optimise system outputs?
- d. Can schema representation and usage in RDBMS systems be adjusted to incorporate more of the theoretical capabilities of axiomatic set theory, particularly the Zermelo-Fraenkel

axiom of the schema of separation? Does such an approach work theoretically for query binding, and can such an approach be implemented in practice?

1.4.2 Research aims

- a. To research the effects of object-relational impedance mismatch and associated factors, such as the impacts of big data that affect relational database query optimisation performance, engaging with the industry practitioner community to research the real-life performance consequences of queries generated from non-traditional sources, including ORM frameworks, upon relational databases.
- b. To identify and develop a novel solution to any adverse performance issues arising from these consequences, testing and validating the solution, and to establish an overarching design framework based upon this solution, detailed at both the theoretical and implementational level, to form the foundation of future work in developing the theoretical bases of this solution further.

The following research objectives are defined to help achieve the aims.

1.4.3 Research objectives

- a. To provide a summary review of the key technical concepts for the topics of this research, and to conduct a topical critical literature review of performance optimisation literature in the relational field together with related topics.
- b. That the literature review in (a) encompasses the evolution of data in information systems; how data has been stored, categorised and measured, with emphasis on the trends and future developments required from data management frameworks to support these expectations.
- c. To investigate and identify weaknesses in current database design and query handling approaches, with particular emphasis on query representation and schema design.
- d. To validate any gaps identified in database performance optimisation research by collecting and analysing qualitative subjective data from industrial practitioners and from academic

professionals.

- e. To identify suitable approaches to developing a conceptual solution to address the identified weaknesses, generalising this solution into a theoretical framework to augment current database storage designs, access methods, management processes and structural conventions, suitable for implementation across platforms.
- f. To investigate if alternative computational optimisation tools and approaches, such as machine learning algorithms, can be used within a solution to the identified performance optimisation problems; if so, to present such a solution design and implementation.
- g. To evaluate the contributions of this research and propose new directions for further work based on the outcomes that were achieved.

1.4 Research Approach

1.4.1 Research philosophy

The focus of the research is on exploring the methodologies and potential performance benefits of a new dynamism in database performance optimisation. To conduct this research, it was necessary to choose a research philosophy which reflected the investigation of untested ideas, and which would be the most effective in answering the research questions, and which allowed for varying modes of enquiry with a selection of mixed methods.

To this end, the philosophical stance of the research is based on pragmatism; this is an approach in which claims to knowledge are made based on actions, situations and consequences [26] rather than as a result solely of strict post-positivistic scientific enquiry strategies, or interpretivist socially-oriented approaches, although the philosophy of pragmatism may encompass both of these. Pragmatism is focused on developing the solutions to problems rather than concentrating on the methods that are used [27], and as such is suited well to a mixed-methods research strategy.

Peirce, cited in Ormerod [28] is credited as one of the principal proponents of pragmatism, and defines it as a ‘philosophy of meaning’, with Ormerod further commenting that utilitarianism has a strong bearing on the meaning of pragmatism. Little mention is made in the literature of pragmatism about specific strategies of inquiry, and pragmatism as an approach appears to be

suited to research where any suitable strategy of inquiry can be considered valid; a disconnection is made from an absolute version of the truth, and the interpretation of truth at different points in time – or, as Melles [29] puts it, ‘... individual action and experience in the world [is] the most realistic basis for decision-making’.

Our solution, while constructed in such a way as to be platform-independent, and with theoretical, set-theoretic and scientific design underpinnings, may be used in the future as a basis for implementation of the ideas within in existing or new RDBMS systems. Finding and testing these ideas using a pragmatic, ‘what works’ approach may then be superior to other research philosophies – for example, the interpretivist approaches associated with social sciences [30, 31], where opinion and narrative are given greater prominence than quantitative empirical testing, may be of limited use when deciding which design approaches provide the most quantitative utility.

Misak [32] argues that under the pragmatic model, beliefs are true for an individual, and the definition of truth is variable according to what ‘needs’ to be believed at the time. This is similar to the importance placed on individuals ‘lived’ experiences in other disciplines. ‘Individual’ could be extended to ‘system’, and this viewpoint can be useful: if this research were to make suppositions, or hypotheses based not just on logical empiricism but based on the humanistic outputs, or relative truths, of the qualitative research, then a richer and more flexible version of the solution might emerge. To illustrate this point, historical research into schema scalability strategies resulted in the concept of normalisation ([33][34]). This is a form of logical empiricism, where the concept can be proved mathematically, and the benefits tested scientifically. Schema normalisation was suited to environments where the variety and structure of information queries was a known quantity (the ‘absolute truth’) but failed to recognise two important factors – the performance costs associated with a decentralised schema [9], and the human difficulty in, for example, identifying functional dependencies that are associated with designing such schemas – Lee [35] noted that ‘...the determination of appropriate normal forms frustrates many systems analysts.’ This discord is evident in the design choices of several current enterprise software packages and has been reflected in comments from our interview participants.

Arguably then, the mathematical and scientific rigour of the normalisation model fell afoul of the dynamic and flexible environmental contexts in which these normalised schemas were to be used, evidenced by the increasing variety of alternative structural approaches [22, 23] to relational databases emerging today.

Floridi [36] defines ‘pragmatic information’ as how much information is carried from informer to recipient, in a specific ‘belief state’, in a specific operational environment. This could be a

considered a comparative philosophical definition of an information system based on interactions (or transactions) such as an RDBMS; a pragmatic approach to the research, then, recognising that truth depends so much on the context of the application and the priorities of the participants (both human and machine), would appear to be an appropriate research philosophy to investigate a pragmatic information system.

1.4.2 Methodological choice

With a mixed-methods approach underpinned by a pragmatic research philosophy, Creswell and Plano Clark [37] argue that the use of both qualitative and quantitative approaches used together in the correct fashion can yield a stronger study than either alone. The mixed-methods approach allows for the human outputs of conversations brought from open-ended interviews to be combined with quantifiable survey outputs from a population of database professionals to identify the primary database performance difficulties experienced in the field; these insights can be used as inputs to the literature review, which can then yield, through an iterative, inductive reasoned approach of triangulation, in a detailed description of advances in the various academic disciplines which contribute to the design (for example database performance tuning; functional dependency identification; sort-merge algorithms; machine learning techniques, for automatic query classification and so on). Further, in designing and testing the functions which comprise the elements of our solution (such as dynamic schema redefinition), the mixed-methods approach allows for the integration of formalised quantitative testing techniques to validate the outcomes [38]. Moving from the outside to the inside of the Saunders' *et al* 'research onion' [39], the following paths, shown in Fig. 1, have been identified to classify the chosen research approaches, each marked with a black rounded rectangle.

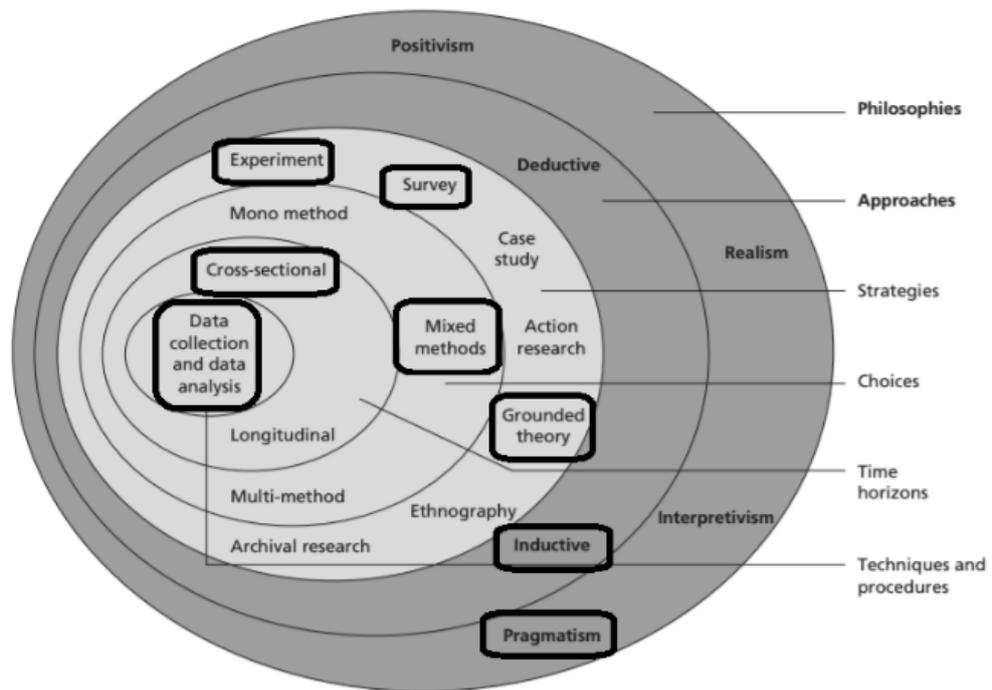


Figure 4.1
The research 'onion'
 Source: © Mark Saunders, Philip Lewis and Adrian Thornhill 2008

Fig. 1.1: The 'Research Onion', adapted from Saunders et al. (2008) [39], with annotations (copyright as shown).

An inductive approach has been chosen for this research. Inductive approaches focus on drawing out the general theory from singular examples, whereas deductive approaches draw singular examples from the general theory. An approach loosely modelled on Glaserian grounded theory is used for the literature review and according to Lapan [40], it is an inductive reasoning method that fits well with qualitative research. Within grounded theory, the Glaserian approach [41] is held to be the most inductive, with the focus being on the integration of findings and letting theory emerge from the data. This also applies to the quantitative testing; where a theory can be tested, it is the singular hypotheses that are confirmed or otherwise by individual experiments that will shape the outcome (i.e., the success or failure of the aspect of the solution under test).

Referring again to Saunders [39], both experiments and surveys have been highlighted, although interviews are also used, and a cross-sectional approach taken – this is to say that research is on the specific, rather than the general case, due to the breadth of the relational model and the range

of testing that could be carried out, once again fitting with an inductive methodology.

Creswell describes six mixed-method design strategies – the ‘sequential exploratory’; the ‘sequential explanatory’; the ‘sequential transformative’; ‘concurrent triangulation’; ‘concurrent nested’ and ‘concurrent transformative’ [37]. The category which best describes this research is the sequential exploratory, where qualitative data collection is completed first followed by quantitative experiment design and data collection to explore an idea.

Finally, the epistemological perspectives are both subjective (for the qualitative research) and objective (for the quantitative research), although it must be noted that a certain rigour is present in the literature review research method that is drawn from an objective, semi-formal design, as detailed in the next section.

1.4.3 Research plan

The research plan is structured as shown in Fig. 1.2.

The plan is split into three phases. Phase 1 focuses on project planning, carrying out secondary research and planning and executing the initial qualitative research and problem validation. Phase 2 looks to define the solution from the outcomes of Phase 1 and in doing so, follows a loose iterative software development methodology, insofar as this can apply to a single participant. The functionality is designed and documented, and in Phase 3, experimental testing and validation takes place, including external validation with academic and industry experts.

Initially, the plan began by researching terms at the highest level of abstraction given the problem domain; database performance optimisation research, with an emphasis on recent developments, with seed terms informed by the author’s industry background. An approach based on Glaserian grounded theory was then used to search the literature, analyse the findings and extrapolate further areas of potential research, by working from general topics, noting the subtopics and findings that emerged and recursively searching and aggregating sources and findings based on the results.

Grounded theory is normally used in social sciences, but is a suitable research method when analysing large, unstructured data inputs such as published literature to find insights regardless of specialty. Grounded theory is a technique for methodologically determining linkages between different data sources, categorising and dissecting the data to find new categorisations and research leads through codification of concepts and comparative analysis. As an inductive reasoning method [40, 41], it fits with the general qualitative research philosophy outlined. To use it effectively,

starting with a general research concept such as database performance optimisation without looking to answer specific questions means the domain remains broad and new insights can emerge. Once a broad and deep study has been conducted, and findings codified with memos outlining key ideas, these can be brought together to form clear conclusions and bring forth theory – for this research, these are indications on which optimisation methods have been popular or successful, and more crucially where gaps in performance optimisation theory remain.

Since the literature review is interaction only with published works and not with human participants, a more objective view can be taken when assessing sources and to this end, a quantitative evaluation method has been used, making this approach *quasi-grounded* theory, rather than a full implementation of the technique.

Database performance optimisation is not the only area which was targeted in the literature review. Additionally, other topics became apparent, particularly when considering other cross-disciplinary techniques. The literature review was expanded beyond the initial scope to include related different topics. In Table 1.3, some potential broad subjects for literature review are related to the research questions. These were eventually refined into the topics that head each subsection in Chapters 2 and 3.

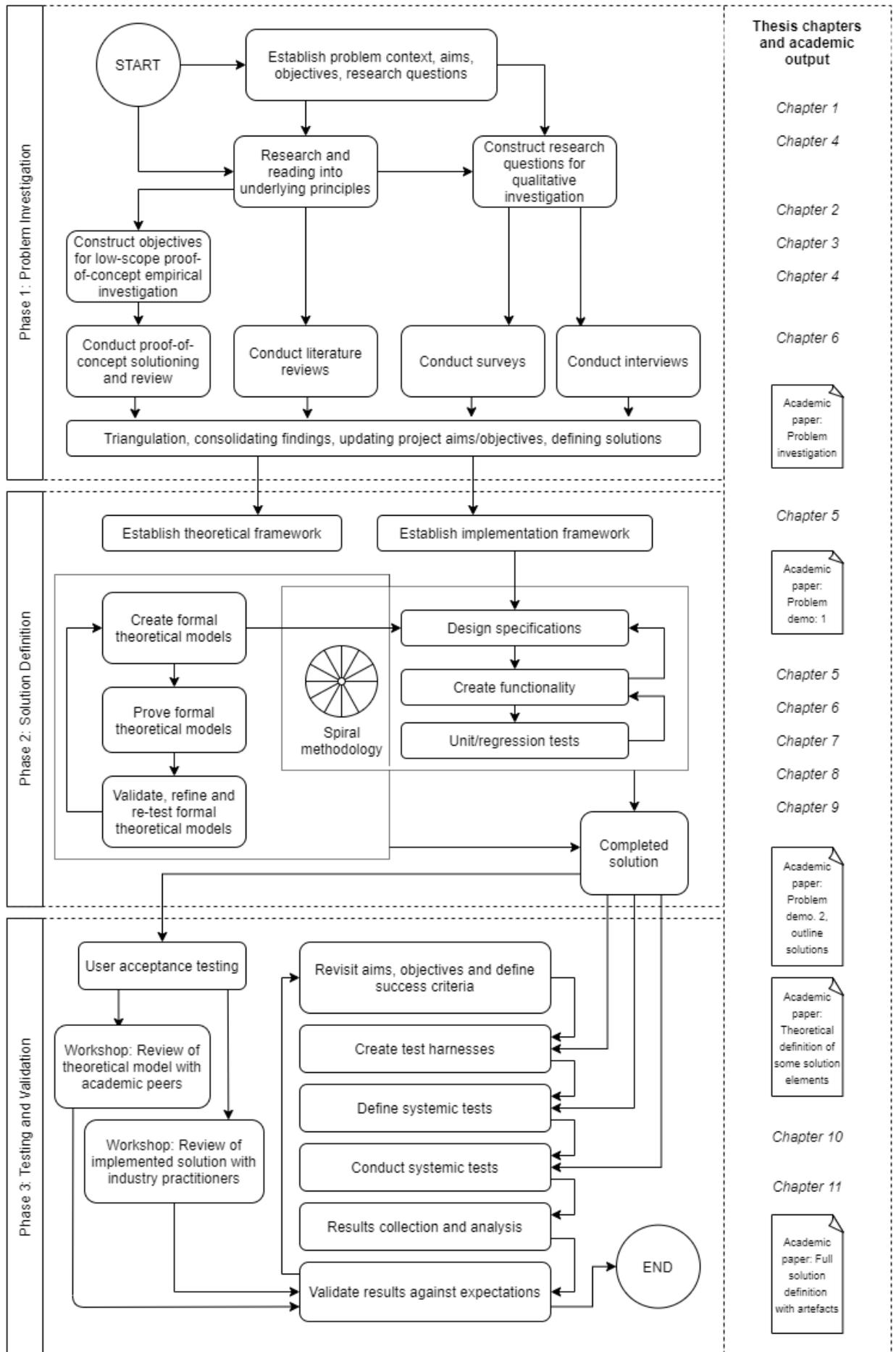


Fig. 1.2: Three-Phase Research Plan

Other approaches were possible for the literature review. One of the criticisms noted [42] is that some vital topics to that study are excluded as they are separate from the interconnected streams of topics identified through grounded theory, and it was necessary to add more ‘seed topics’ (see diagram above) to establish a broad view. Additionally, with increased review into secondary and tertiary references, the scope of material necessarily extended backwards in time, meaning topics quickly became outdated or irrelevant to the primary study.

Table 1.3: A sample of topics related to Research Questions A through D

| Research Question | Topics |
|--------------------------|---|
| a, b, c, d | Database performance optimisation. |
| A | Data evolution; data culture; big data; unstructured data; Agile; object-oriented programming; Internet of Things; application design methodologies; data warehousing; distributed data. |
| b, d | Graph theory; multi-dimensional information representation; Hilbert spaces; matrix theory; linear algebra; machine learning techniques; code refactoring; learning algorithms; artificial intelligence. |
| C | Measuring data; 3 ‘V’s; domains; statistical modelling; planes / complex planes; data classification; pattern matching; data aggregation; data types. |

Examining current research only helps avoid the latter point, but also excludes certain vital necessities – such, as in this case, vital work on set-theoretic constructs, like domains [43]. Fig. 1.4 illustrates the approach used to discover, assess, analyse, codify and extract meaning from research literature using the grounded theory approach.

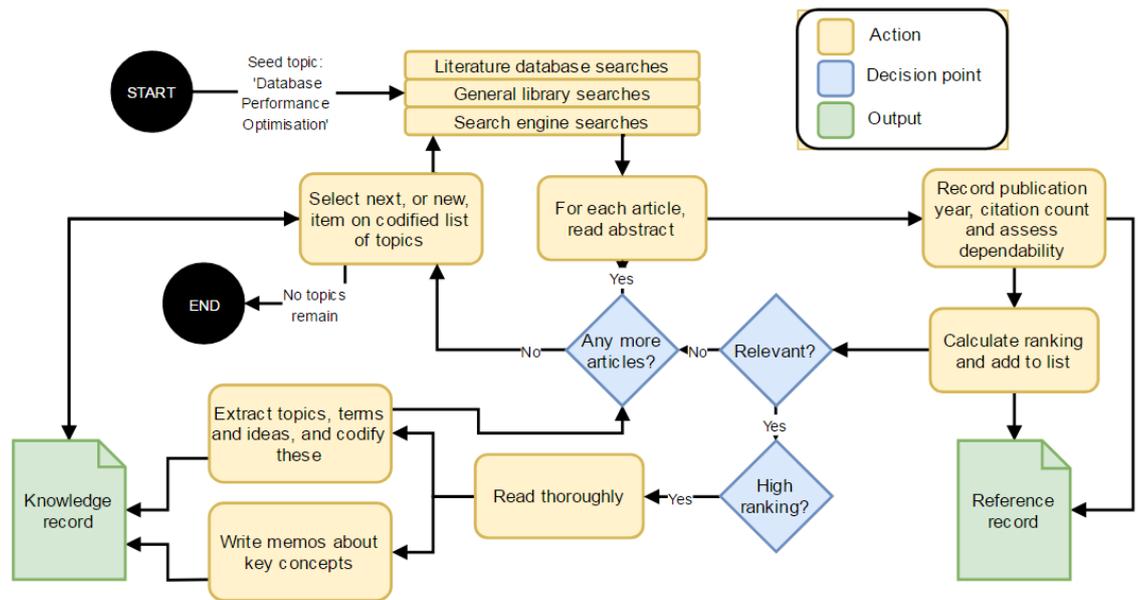


Fig. 1.4: The grounded theory-based approach to literature review

1.4.4 Primary qualitative research strategy

One purpose of the secondary research is to discover the limitations and weaknesses in current database optimisation research, implementation methods and best practices. While a significant amount was discovered from examining sources such as academic journals and technical documentation, much of the knowledge pertaining to efficient database performance tuning practices is latent and highly dependent on factors such as environment, experience of the professional, corporate policies, personality, role, software version and business context.

For this reason, making assumptions about the limitations of performance optimisation would be best augmented by consulting established professionals to gather more detail on alternative viewpoints to the problem which will help to establish the scope and priorities for the solution. This is an example of the mixed-methods approach to the research, where using interpretivism (as part of a pragmatic approach) alongside a more structured quantified positivist style can result in a survey where answers can both be counted and interpreted to form conclusions.

The surveys took the form of a mixture of multiple-choice questions, and open survey questions. The target audience for the first survey were professionals engaged in active positions requiring interaction with database management systems. These included developers, database administrators, analysts, academics, IT managers, application support specialists, architects and so

on. Due to the delivery medium (online, group targeting) it was not possible to strictly filter out other professions; however, the distribution of the survey was targeted to those communities most likely to have members engaged in these professions and their primary occupations were captured in the survey, so that non-qualifying participants can be excluded in analysis.

As this is an inductive method, no prior hypotheses are assumed. To facilitate this, the questions were designed to be balanced and without lead or bias. The first survey was split into three sections –

- The profiling section, where the respondents were invited to provide some background information. Personally-sensitive information such as gender or race were irrelevant for the survey analysis and so do not need to be recorded, but data points such as number of years of experience and job role were captured here. These responses also qualified the respondents to answer the main questions. Given the desired target audience of regular database users, developers, administrators etc., an early exit point was built into the design in case respondents do not have sufficient regular experience to assure the desired competency and experience in the field.
- The next section was focused on the processes, procedures, tools and frameworks that the respondents were currently or recently using. This formed a snapshot of their current opinions and methodologies and these questions were designed to provide information to help answer the first research question.
- The final section, where questions were focused on potential improvements that the respondents have planned or would like to see. This section looked at what is possible in the area, and what changes the respondents would make. The outputs of these questions helped inform the design outcomes specified by the remaining research questions.

To build upon the outcomes of the questionnaire, three semi-structured interviews with leading database professionals were undertaken to collect opinions on both the current performance tuning challenges and future directions for database performance research and implementations. These interviews produced insights which complemented the outcomes of the literature review and survey(s) to determine the best possible design paradigms, and these outcomes are presented in Chapter 4.

In keeping with the inductive reasoning approach, these interviews were narrative, in-depth interviews conducted on loose lines of enquiry. Taylor et al [44] note that this style of interview is,

‘... modelled after a conversation between equals rather than a formal question-and-answer exchange.’ This is a style where rapport is established between the participants and non-directed conversation occurs to bring out opinions and other data for later analysis.

Interview audio was recorded in full and transcribed for analysis. Information analysis was conducted through extraction of opinions and ideas expressed by the interviewees using first-pass sentiment analysis with the software package NVivo and analysis and final codification by hand of the findings, and the categorisation of these, alongside the survey output, into short conclusions and directives that later informed the solution.

1.4.5 Primary quantitative research strategy

The research strategy first focused on establishing the scope of the problem in the field of database query representation, parsing and re-use strategies, doing so using secondary research in the form of literature review and primary research through problem validation with industry professionals and academic experts.

Beyond this initial strategy, the research branched into the quantitative – establishing a base design constructed from our findings and establishing an initial proof-of-concept. This proof-of-concept was designed to test several of the key tenets of any potential solution for feasibility, and to help answer research objectives (e) and (f), before proceeding to develop a full theoretical and practical solution:

It was established through the qualitative research that a new approach was needed to address the following findings; this informed our initial high-level solution design using a top-down design approach in a spiral methodology based on the following findings from the qualitative research:

- That external application usage patterns have changed, particularly with the advent of ORMs, resulting in changes to query patterns that are not well served by the static nature of current query parsing and recaching methodologies in RDBMS systems.
- That a more efficient approach is desirable that improves query re-use and overcomes caching issues.

- That there is an opportunity to use certain elements of set theory to introduce dynamic schema creation and selection into RDBMS platforms as an additional strategy to help address poor query handling.

An initial sample data set from the public domain was then identified that could be split into sub-schemas; SQL queries to address this data set were created; and an initial implementation for the query representation and schema selection elements of our high-level design was created, drafting these both in theory and implementing in practice, using PostgreSQL as the RDBMS and Python 2.7 for the new feature code. Test harnesses were built and several hundred tests executed, with the results recorded for later quantitative analysis.

The following methodological choices were made:

- Staged, modularised development using a spiral methodology was chosen as the most apt approach to the solution under investigation. In this methodology, requirements gathering, design, implementation, testing and deployment are arranged in concentric spirals with each journey around the spiral encompassing more requirements and consequent features in the final artefact. This methodology can be managed by a single researcher and any definition of ‘done’ can be defined, however it has the disadvantage that the single researcher will by necessity need to fulfil all roles in the spiral.

Fig. 1.5 illustrates the original concept of the spiral software development methodology as developed by Boehm [45]. Although some phases such as risk analysis are not relevant to this investigation, the overall concept of moving between 4 key phases and developing an increasingly complex artefact based on frequent review remains.

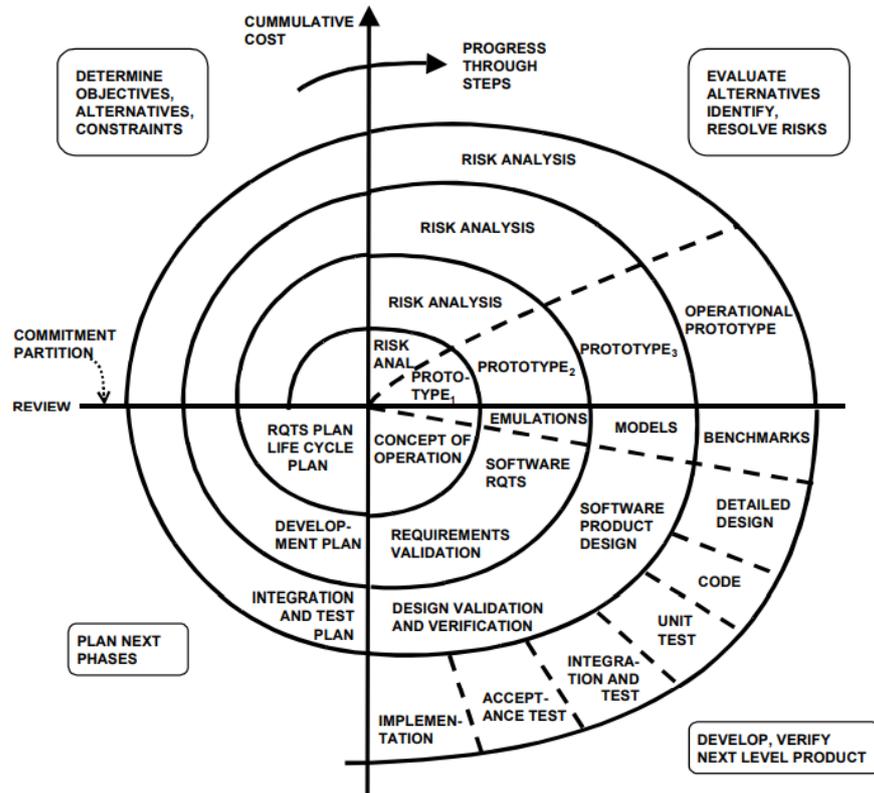


Fig. 1.5: The original spiral methodology, reproduced from Boehm, 1988 [45, pp.64].

The following methodologies were considered and rejected:

- Waterfall (BDUF) methodology: Since the aim is to establish the feasibility, through design and experimentation, of a pragmatic solution to the research problem, a waterfall methodology would provide too much rigidity between requirements gathering, solution architecture, implementation and testing. Since trial and error is required, an iterative approach is more appropriate.
- Agile methodology: This methodology is better suited to teams where the goals can be split into a series of tasks, grouped into sprints and allocated to individuals or small teams. Each task is tangible and has a clearly defined definition of ‘done’. This methodology was rejected (and variants of it, including XP and Scrum) as over-engineered for the purposes of a project involving a single author.
- V-model methodology: This model relies on the pairing of planning and development tasks with testing and deployment tasks in a series of roundabout interactions to ensure that requirements are validated before design proceeds; that design is verified before implementation proceeds; that implementation is tested before deployment proceeds. This model relies on some division between the

development and the testing; some pre-determined knowledge of the eventual design, in the style of waterfall, is required, and this approach often involves multiple teams. This approach was rejected in favour of the spiral methodology which enables iterative solution development without fixed definitions of the final artefacts.

The test harnesses, test definition and execution of the initial proof-of-concept was conducted using a mixture of quantitative approaches. For the initial proof-of-concept, the scientific method was used, with hypotheses defined and outcomes compared against the hypotheses. For later development and testing, a more exploratory approach was used, with goals defined for the functional units of software and tests defined to establish whether the goals have been met; a commonplace approach in software development. A range of statistical methods have been used to establish success, including a standard range of statistical aggregates, analysis of p-values using T-tests where necessary; formalisation of the theory underpinning the solutions as algebraic expressions, using both set theoretic notation and the relational algebra; and discovery and display of our results using a range of graphs and visual metrics.

1.5 Ethical Issues

Some constraints were in place for the interviews. Many professionals in the field are based in the countries in which the main platform providers primarily operate; thus, the interviews were conducted via video conferencing technology for reasons of economy. This was convenient, but to a certain degree removed some of the interpersonal rapport between the interviewer and interviewee that may otherwise have elicited more detailed, honest and comprehensive responses.

Aside from ordinary ethical precautions associated with conducting interviews and data handling requirements dictated by the Data Protection Act (2018) and latterly the replacement General Data Protection Regulations (GDPR) when dealing with personal information, there are no other ethical concerns related to this research. All third-party respondents are non-vulnerable adults participating voluntarily and knowingly in activities that are solely verbal or electronically interactive, dealing with non-sensitive topics.

Proportional ethical approval for both the survey(s) and interviews in the formats described was obtained from the Principal Supervisor and the Faculty of Computing, Engineering and Sciences Research Ethics Committee on 24 March 2017.

1.6 Thesis Structure

This chapter has provided an introduction to the research and specified the research questions, aims, and objectives; outlined the contribution to knowledge, and shown which research philosophies, approaches and tools were used to carry out our investigations. Research outcomes and ethical considerations have also been discussed.

RDBMSs and the relational model more generally have a long and detailed history. Chapter 2 (Background) describes the theoretical underpinnings by discussing previous contributions to this history from many of the seminal authors and practitioners in the field. Several of the issues in relational theory and practice are defined and linked to some central causes - the increase in the volume, variety, and velocity of data at scale; the emergence of Object-Relational Mapping (ORM) frameworks, their associated performance anti-patterns and the extent to which ORMs have been embedded into software development and release architectures, and a discussion on their contemporary applications.

Chapter 3 (Literature Review) presents a selected topical literature review; these topics include the challenges presented by ORM platforms; relational query performance optimisation; the effects of the 3 'V's of big data; and advances in alternative query representation forms. Object-relational impedance mismatch is described and defined in detail and compared to prior literature which has aimed to solve or mitigate the resultant practical issues. This section also examines the difficulties in performance tuning queries posed by over- and under-normalisation of schema architecture and presents some of the extensive literature in this area.

In Chapter 4 (Problem Investigation), these issues are specified more clearly, addressing their scope and applicability to the aims, and this section presents our primary qualitative research, a series of surveys conducted both on an individual level through expert interviews and through engagement with the wider technical community in the form of a tailored and targeted questionnaire. The findings are augmented with references to the findings of the published papers that emerged from this research, which focus on the issues created by ORM products in RDBMS engines and include practical demonstrations of these shortcomings in both theory and practice, using current enterprise tooling. Two separate real-world datasets and two different RDBMS platforms are used.

In Chapter 5 (Solution Design), having established the background and current state of the literature in Chapters 2 and 3, and establishing the depth and scope of the problems in Chapter 4, an overarching solution is defined. This solution, termed PETAS (PErformance Tuning with Adaptive Schemas), is comprised of several elements, each element working together to provide an alternative methodology for query handling, caching and execution. This section expands upon a key deficiency in the very kernel of the RDBMS engine, particularly in how SQL queries are parsed, cached and optimised – a deficiency which is common across RDBMSs. This section presents an argument showing how this fault stems from an internal query representation problem and proposes a new method for internal query representation, the multidimensional adjacency matrix. It is also shown how queries can be compared and ranked by using this matrix method combined with Hamming distances and the use of a statistical technique (k-nearest-neighbour) more commonly associated with machine learning.

Chapter 6 (Testing: Query Representation) is a deconstruction of the first element of PETAS, the novel query representation in graphical form using multi-dimensional adjacency matrices. Continuing from the solution description in Chapter 5, a brief introduction is provided and an implementation of this component is presented. This section details the experimental testing details the research outcomes. The applicability of this approach to RDBMS systems in general is discussed.

Chapter 7 (Testing: Schema Selection) extends the description of the schema selection mechanism from Chapter 5, which makes use of some simple machine learning algorithms to classify inbound queries as belonging to certain pre-defined schemata. A working k-nearest neighbour implementation is demonstrated and tested alongside the query parser. This section provides evidence of a working implementation and documents the results.

Chapter 8 (Testing: Dynamic Schema Redefinition) describes the third innovation of PETAS, the dynamic schema redefinition mechanism, which leverages the principles of axiomatic set theory to allow RDBMS systems to maintain multiple, parallel schemata which are simple transformations, translations and subsets of a base schema. Leading off from the definition of this component in Chapter 5 and in particular the novel definition of query efficiency, it is shown how this feature can be used independently to provide both tangibly faster query execution for a variety of query types. This section provides evidence of a working implementation using materialised views as substitutes for alternative table metadata definitions and indicates how this can be used alongside the query parser and selection mechanism.

Chapter 9 (Conclusions, Reflections and Future Work), summarises the overall validation, testing and results of the experimentation on each component of PETAS. This chapter brings together the previous qualitative work with domain experts, the quantitative testing carried out using empirical methods, and the testing of each component as described in Chapters 6, 7 and 8. The outcomes of testing our PETAS implementations are demonstrated, integration testing is discussed, and the strengths and weaknesses of our testing methodology and overall solution are considered. This section brings together all the strands of the research and presents the conclusions, revisiting the aims and objectives, considering the novel contribution to research and summarising areas for future research that can develop these ideas further.

The Appendices, containing supplementary material as directed throughout this document are included at the rear.

1.7 Novel Contributions to Knowledge

The novel contributions to knowledge that this research provides are summarised as follows:

- The research and production of a novel query representation technique to store database queries as multidimensional adjacency matrices – directed graphs in an array form.
- The research and production of a novel algorithm for similarity scoring, using existing techniques but applied to multidimensional adjacency matrices in such a way as to effectively compare the structure of any two matrices and produce a normalised linear output.
- The research and production of a schema mapper component which can effectively assess inbound queries, adjust internal weights and rank-order queries by relative accuracy in predicting performant sub-schemas.
- The production of a method for subset schema generation through dynamic schema redefinition – while this element in particular is based upon existing methods such as materialised views, and similar ideas have been explored before [6], this method is novel in the interface with the schema mapper, the definition of a new efficiency metric and the definition of a view variant which accesses data pages directly without reference to a base schema, a deviation from the traditional view.

- The publication of three conference papers and a journal paper which detail the problem investigation and the different components of PETAS.

1.8 Chapter Summary

This chapter introduced the research project, stated the project motivation, and made the case for the importance of investigating the effects and solutions to the impact of object-relational impedance mismatch upon relational database management query optimisers. The chapter specified the research questions, aims and objectives, and commented upon the novel contributions to knowledge, narrowing down two key deliverables. The research approach was identified and, using Saunders, the paradigms, approaches and techniques were chosen. This section also presented the three-phase research plan and detailed the selected approaches to the qualitative and quantitative aspects of the research methods for both the primary and secondary aspects of the research outcomes. Finally, ethical issues were summarised.

Chapter 2 provides a comprehensive overview of the problem background incorporating a summative literature review, expanded into a topical in-depth literature review in Chapter 3.

Chapter 2: Background

2.1 Introduction

This chapter discusses the fundamental ideas intrinsic to the remainder of this research and defines the key terms, providing a summary literature review on query performance concerns in relational database systems. The Relational Database Management System (RDBMS) is defined and explored; this chapter elaborates the definition of a database query, the underlying data structures are characterised, and the accompanying Structured Query Language (SQL) used in RDBMSs is described. The RDBMS and the SQL language are linked by outlining the query optimisation and execution process inherent in RDBMS systems, and the life of a query from inception to completion is illustrated in Section 2.4. Some issues around query performance tuning are examined, particularly with reference to the difficulties of tuning a query and the accompanying effects this may have on application dependencies; some other strategies for schema-driven performance tuning are also described. Finally, the summary brings together the key points from this chapter.

2.2 Relational Database Management Systems

2.2.1 Overview

Data is all-pervasive, and intrinsic to almost every interaction one has with the world. Increasingly, people are choosing to measure, store and interact with data through, for example, consumer home automation devices, and self-management of fitness and wellbeing with wearable devices [1]. As society takes an increasingly interactive role with Internet-connected machines, the data that these devices generate must be stored safely, securely, consistently and must be available when needed. In this sense, and like the traditional data collated and held by organisations, improving the ability to optimise and improve the performance of relational database queries across a range of database management systems remains a central issue in computer science [2, 3].

In relational database theory, a schema is a collection of database objects, primarily but not only tables, with each table being a collection of data points organised into columns and rows [4, 5, 6]. Such an arrangement is designed on the principles of relational algebra from axiomatic set theory [7], and tables are described as relations, although there are differing opinions on this definition [8]. Database queries are used to access this relational data through a relational database management system (RDBMS) interface.

Databases may have multiple schemas, and each schema represents a collection of tables or relations which correspond to one single physical arrangement of data in the storage layer [9]. By physical arrangement, it is meant that the data itself is arranged into a structure, typically a series of pages. Each page is a fixed size, and the data is stored within these pages in an ordered manner, with each page having a physical address in the storage layer, with collections of pages supplemented by metadata, such as index pages, describing their structure [10]. It follows that the table structures and consequently the schema structures as implemented in the RDBMS are therefore logical constructions since the schema is a logical abstraction of the collection of storage addresses.

The tables within a schema remain static in contrast to database queries which are flexible in structure, easily changed, and able to project, join and filter data from a variety of tables to meet the users' requirements, bounded only by the confines of the objects present and the query language dialect in use, both dictated by the tenets of the relational model. The static nature of the relational schema is arguably a disadvantage in database systems with much flux in the variety and volume of the data, as noted by Nayak et al. [11] in comparison to the general class of NoSQL systems which provide a wider range of data models.

The primary language for interacting with relational database systems is Structured Query Language (SQL). There are several dialects available depending on the implementation of the RDBMS, but the core operations are defined within a set of standards, the latest iteration of which is ISO/IEC 9075:2016 [12]. SQL itself is split into several sublanguages, and in RDBMS systems, the two key divisions are Data Manipulation Language (DML) and Data Definition Language (DDL). The former is responsible for creation, amendment, and destruction of database objects (using commands like CREATE TABLE) and the latter is responsible for the aforementioned data-centric operations (such as INSERT INTO [table]).

Database queries are implementations of the relational algebra, a set-based logical method of arranging data into domains and sets of related domains, and the methods for operating upon these sets by projecting, aggregating, matching, and filtering these sets into subsets. Therefore, query performance tuning – specifically the methodology of presenting queries to RDBMSs in such a form so that they execute efficiently – is a problem that can be abstracted from query languages to the relational algebra. Since each database query has a relational algebraic representation, and that the processing of a query can be described as the application of several algorithms to the query, it follows that the act of processing a query can be reduced to a description of the application of a set of logically- or mathematically-described algorithms to a set-theoretic algebraic expression; thus query processing is generalisable from the specific SQL case to the abstract logical and algebraic case. This is not always true since SQL extends the relational model; likewise, there are some operations in the relational model, such as relational division and renaming, which are only

indirectly supported in SQL.

2.2.2 *The role of the schema*

One well-understood categorisation of data access approaches is the so-called schema-on-write and schema-on-read separation, where in the former case, the schema, or data structure, is already known and the data is written into this structure. In the latter case, the data can be unstructured and the data is simply written as-is, with a schema (if needed) being defined whenever the data is retrieved [13]. The former case, schema-on-write, underpins the fundamental design of relational database management systems (RDBMSs) [3]. An RDBMS is designed to store data in predefined schemata (plural of schema). Unlike static data stores, relational systems have the advantages of being able to incorporate key set-theoretic ideas, such as the idea of selecting combinations, intersections or aggregates of different data from the tables on-demand, and being able to select, filter and arrange the data to suit [4]. Data can also be inserted, updated or deleted according to set criteria and manipulated *en masse*. The functional programming language SQL (Structured Query Language) is a common and widely-implemented method of constructing queries to do this - queries are sets of these commands [14].

2.2.3 *Current issues*

Today's RDBMS has other functions alongside data storage and retrieval. It must provide the capability to store the data in a confidential manner, ensuring integrity, and make the data available when required. These attributes, also called the C.I.A. (confidentiality, availability and integrity) principles, have been long understood as core components of information systems [15, 16]. Today's users of database systems also demand other attributes such as high availability, the ability of the database system to withstand disruptive, availability-affecting events such as power outages through techniques like redundancy; interoperability, the ability for the database system to interact seamlessly with other technologies such as object-oriented programming languages and non-relational data stores [17, 18]; and business intelligence, the capability for the database platform to integrate and support data visualisation and analysis by end-users.

The divergence from the traditional definitions and limitations of a database characterised by a plethora of old and new implementations presents issues when trying to maintain standardised interfaces, as seen by the addition of platform-specific functionality per implementation of the SQL language. In addition, as Agarwal et al. [19] note, data is resident in many types of platform, not

just the divergence of relational platforms, and integrating this data remains a current and significant challenge.

Other challenges present themselves. The movement from functional to object-oriented program development techniques over the last 50 years has led to an increased awareness of object-relational impedance mismatch, where the object-driven methodologies of application development collide with the set-driven, functional paradigms of relational data [20], which give rise to new performance challenges for the RDBMS. The explosion in so-called 'big data', data which is characterised by high velocity, variety and volume, amongst other 'Vs' [21], can be a test for the scalability of RDBMS solutions.

2.3 Query Representation and Comparison

2.3.1 The role of the relational algebra

Database queries are enquiries made upon set-based data structures [6, 7]. Queries are the implementation of a collection of different operations on a data set that can be combined and modified to produce the required results [5].

The fundamental operation in relational algebra is the projection. This can be defined as some restriction of a set of tuples (set of related values) and a restriction across a set of attributes in a domain. Another way of describing this is a subset of any larger set, where the subset consists of some related values across a equal-or-larger set of possible values, and where each value is a member of some wider possible range, or domain. This is implemented in SQL as a SELECT statement. Note that the subset can be the whole set (or more formally, the cardinality of the subset is equal to the cardinality of the whole set, also known as an improper subset), or simply a partial subset (a proper subset), or just a single value [7].

Other operations include the join, where a target set is defined as some combination of one or more disparate sets, also known as a composite relation [22]. For example, an inner join, or in set parlance a theta-join or an intersection, can define a target set combining two smaller sets such as the set of customers and the set of sales. Combining these sets yields advantages in exposing hidden data, such as selecting (projecting) a result set that includes the amount spent per sale per customer. Moreover, different types of join such as an outer join (semi-join) can be implemented in database systems. Notably, the anti-join (a join between two relations R and S where there is no commonality between the two sets on their join conditions - attribute names) is not implemented in

SQL, and instead is normally achieved through the combination of a semi-join (left join) and predicates (where clauses). Fig. 2.1 illustrates a composite relation (inner join, or theta-join), across three tabular relations.

More set-based operations include the insert, update and delete operations [5], and operations that aggregate information (for example, the sum of sales per customer). RDBMSs are capable of providing query languages that handle most set-based operations but set-based operations are not identical to database query languages. Some operations in RDBMSs are not theorised in set algebra (such as pattern matching with IN and LIKE, although arguably IN can be treated as a subquery, or in set theory, a composite relation that includes another relational expression, and LIKE as a string comparison that includes wildcards).

2.3.2 Query representation

Database queries are the implementations of operations in relational algebra, as demonstrated by many researchers, notably the seminal authors of the language Astrahan et al. [23]; later, Ceri and Gottlob [25] who built a translator showing how SQL queries can be mapped to relational algebra; and Date [8, 25] in various publications. However, SQL can also extend relational algebra, enabling the use of non-relational techniques such as inline functions.

Queries themselves are generated in several ways. The first is manual creation by a developer. This is where the SQL query is encoded into the application within a method call. Typically, this approach is used in older applications where the application code is not expected to significantly change over time. Coding in this manner has some disadvantages - many application development languages are object-oriented, whereas SQL is a functional language. This presents practical difficulties when performing operations like passing parameters from a method to a SQL query, since the query itself is implemented as a text string upon which the parameters must be substituted in as string literals. The problem of incompatibility between functional SQL and object-oriented programming paradigms is known as object-relational impedance mismatch. Ireland et al. [20] identified several layers of mismatch, and the whole topic is discussed more fully elsewhere (since the unique challenges that arise by using mapping solutions to overcome this mismatch are a major driver of this research).

```
SELECT StationName, TemperatureC FROM WeatherStation INNER JOIN WeatherReading ON WeatherStation.StationID = WeatherReading.StationID WHERE ReadingDateTime BETWEEN '2018-01-01' AND '2018-12-31';
```

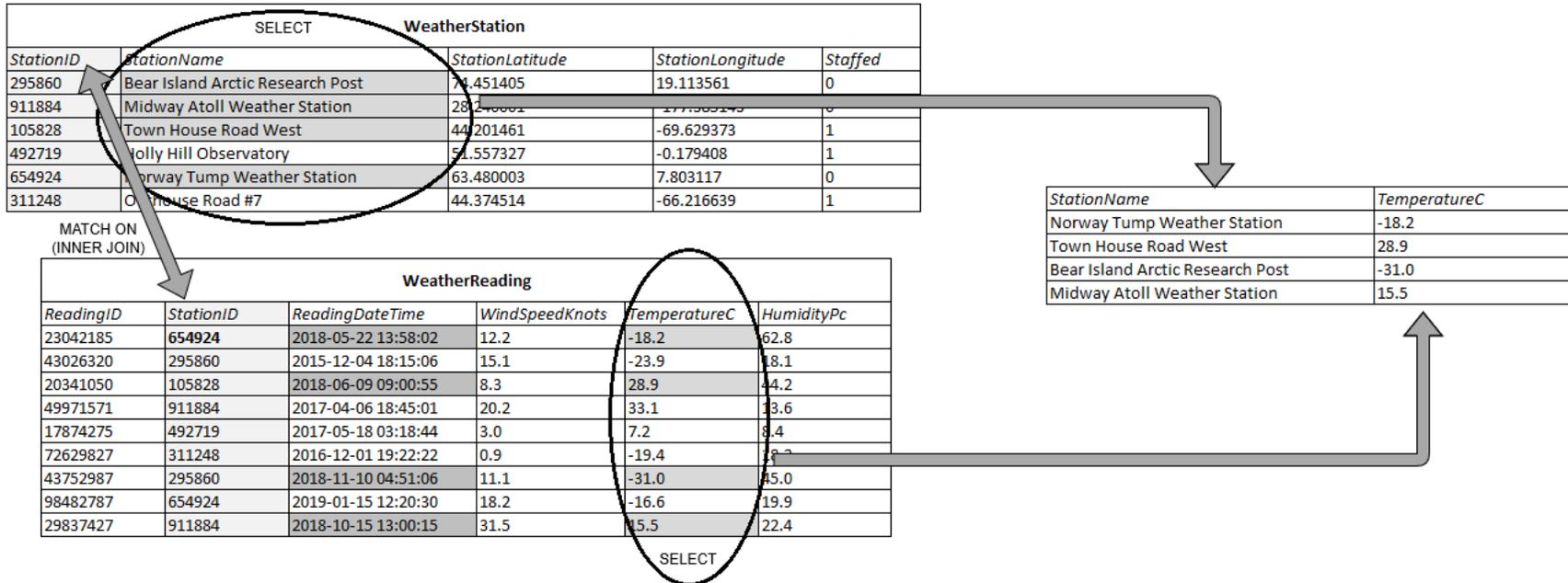


Fig. 2.1: Illustration of a theta-join

Other methods of using SQL queries (which are extensible implementations of relational algebraic expressions) are through stored procedures and functions - pre-written queries accessible within the RDBMS by calling an associated method name. This approach has advantages such as better performance through query caching and better alignment with the object-oriented model, but disadvantages include the overhead of maintaining a separate codebase within the RDBMS and phenomena such as performance issues that result from execution plans derived from poor parameterisation [26, 27]. The third, and increasingly common method of generating queries, is through the automatic generation of SQL that results from an intermediary Object-Relational Mapping (ORM) tool. These are object-friendly interfaces that map method calls to SQL queries opaquely, that aim to reduce the impact of object-relational impedance mismatch so that the application developer writes no SQL but instead calls a method which generates the necessary SQL for the desired operation. Implementations include Entity Framework and Hibernate. While ORMs provide advantages such as abstraction and ease of use, disadvantages include the exhibition of performance anti-patterns [28, 29, 30]. The advantages and disadvantages of ORM tools are described in further detail in the next chapter.

In terms of internal representation, SQL queries undergo a particular process of parsing, binding or algebrisation, optimisation and execution. However, an important part of query performance tuning is the ability for the RDBMS to recognise queries which are similar, or identical, to queries which it has processed before. Each query that is presented to the optimiser results in an execution plan for the query, which is a set of tangible algorithmic steps that can be taken by the database engine to execute the query and return the results. The ability to identify similar queries yields advantages such as re-use of a previously-generated execution plan [4], lowering the time taken to process the query, and the ability to cache the intermediary objects such as the parse tree which reduces the space required for the plan metadata in memory, increasing memory capacity for other queries.

In some implementations, queries can be prepared. The process of preparing queries means to identify the parameters within the query and remove them to a separate list of key-value parameter pairs, to be substituted into the query at run-time. There are advantages to this approach including query re-use and interoperability with wider data processing platforms such as LINQ [31]. However, this approach is dependent on the implementation of the RDBMS. Queries which are frequent often refer to objects which have their pages stored in a buffer cache, meaning a large portion of data retrieval can take place in-memory without reference to the disk subsystem, significantly reducing I/O costs and reducing the query execution time to the advantage of the user [32].

Checks for query similarity are limited by nature, because the query execution process by necessity must be extremely swift and so an excessive level of query pre-processing would impact overall query execution time. For example, in Microsoft SQL Server, there are basic parameterisation options (known as 'simple' and 'forced' parameterisation) which can be selected automatically or overridden by the user [33] which will enable the optimiser to recognise that a query has been presented before, even if certain parameters of the query are different. However, a query which is logically identical but syntactically different (even to the point of having only additional whitespace as the only differentiator) can still be treated as a different query; a major disadvantage for query performance, reducing the efficiency of the query optimisation process. Overcoming this issue through the investigation and implementation of a computational method for internal query representation is a major objective of this research.

2.4 Query Representation and Execution

2.4.2 Query representation and execution

Database queries are semantic structures that use a finite and defined syntax. Whereas data collections in RDBMSs are organised typically into rows and columns, each field (intersection of a row and column) containing a data value, and the whole contained in tables, queries are algorithmic descriptions of operations upon those collections, and have a different structure. In RDBMSs, this structure must be quantified and turned into a set of executable instructions, and this is done via a compiler, in the same fashion as high-level languages are compiled (or interpreted) into a set of machine-readable instructions [4]. Regardless of the implementation details, the steps for compilation and execution of database queries are generally universal.

Fig. 2.2 illustrates the general query execution process followed by most RDBMSs.

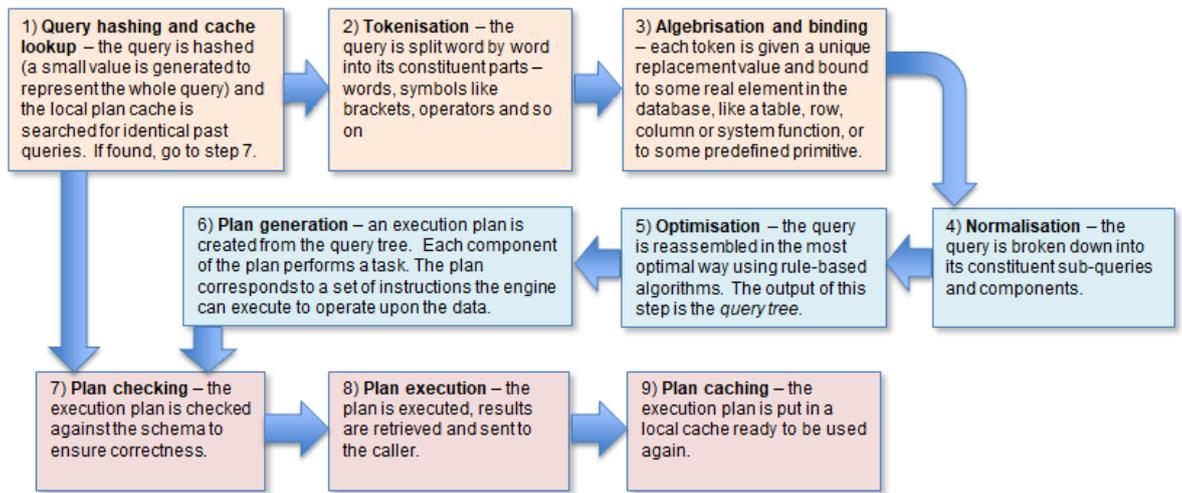


Fig. 2.2: The query execution cycle. Derived from Delaney [40].

First, the query is parsed for correct syntax, and this involves a process of tokenisation - words in the query are delimited and metadata values (labels, or tokens) assigned to each word. Tokens are then grouped together and mapped to internal operations in the same manner as language parsing in natural language processing [34, 35].

Next, the binding (or algebrisation) process associates each token - that is, word or relevant syntactical symbol - with an operation or database object. The outcome of this stage (depending on implementation) is the parse tree or the bind tree (these can be distinct, but not always), which turns the list of tokens into a flow that fully describes the operations and their interdependencies. This is stored in a tree format which can then be read by the next stage of the query optimisation process. Parse trees are described more fully in the next chapter.

In the next stage, optimisation of the bind tree takes place to produce an execution plan [36, 37]. The execution plan is a set of instructions for the RDBMS to execute which will produce the result set specified by the query (or implement the set of operations that the query specified). However, the steps in the execution plan depend upon various factors. The first factor is the nature of the data structures which are being queried, and the operations available to read the data. For example, some RDBMSs differentiate between index seeks and index scans, where seeks look for a specific range of data by traversing to the partition or segment of the data that contains those values (much like looking up a name alphabetically in a phone directory) and scans read the whole table until the appropriate conditions are met i.e. the data is found [38]. Scans are much more costly than seeks [36] due to the additional I/O and CPU load demands, and consequently can take longer to execute, to the detriment of the query being run and the user waiting upon the results.

Other factors include whether indexes are defined on the table structures, as depending on the nature of the query the use of an index instead (an index being either a physical arrangement of data pages on disk or a secondary structure containing the tabular data in an alternative order) can have performance benefits [4, 39] as indexes can be specified as trees and tree traversal can be a highly efficient operation. A third factor is heuristics. Some RDBMSs use these 'rules of thumb' to make swift decisions about the best execution plan for a particular query, depending on the structure of the query [36]. An example of this is where the optimiser decides to use an inner loop join (as opposed to an inner hash or merge join) for two tables as one table has a very low cardinality (population count) and the loop-based operation would be quicker than the pre-sort required for the merge or the bucketisation processes involved in hashing. Implementations differ, and thus execution plans also differ depending on the internal query optimisation algorithms of the RDBMS. Fig. 2.3 illustrates an execution plan, showing the execution plan for an identical query on identical table structures on different implementations (Microsoft SQL Server and IBM DB2). Note how the left-side plan (MS SQL Server) uses a merge JOIN to integrate two sets of pre-sorted results from the table scans, but the IBM DB2 (right-side plan) optimiser chooses not to pre-sort but instead uses a hash JOIN, sorting once after the JOIN is executed, illustrating how RDBMSs can differ in their approach to query optimisation.

Once the execution plan has been produced, the query is executed. This happens through the RDBMS reading the execution plan in the order specified and executing the instructions. Many RDBMSs support parallelism, for example the Microsoft SQL Server RDBMS implementation supports parallelism at the CPU (socket) level, the core level, and the thread level [41]. This means operations on two 'branches' of the execution plan can be executed simultaneously on different processor schedulers, or the workload of a single plan component can be split across multiple system resources (such as CPU cores). An instruction such as 'index scan' may involve the RDBMS accessing index pages on disk which specify where the data sought can be found (using page addresses and offsets).

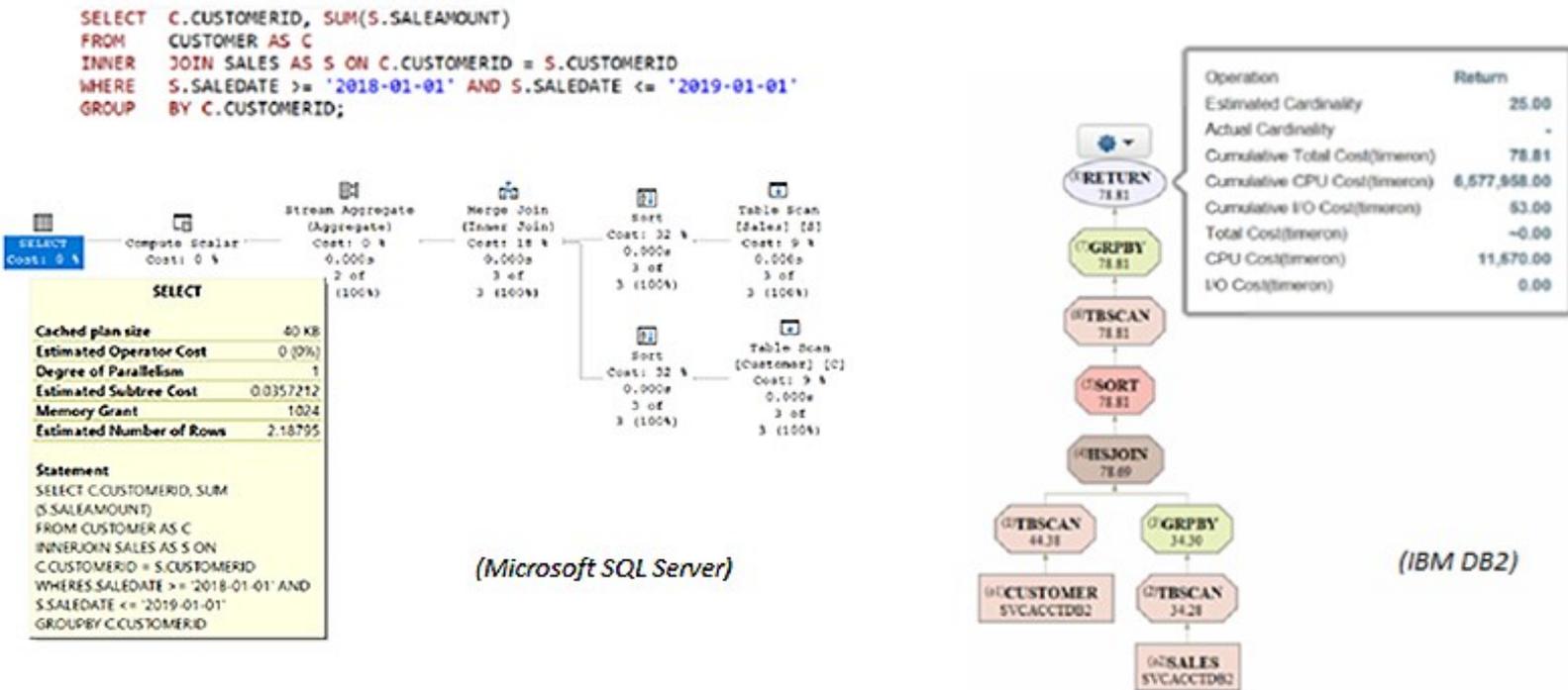


Fig. 2.3: Two execution plans compared

This data is then read by the I/O subsystem in the OS, written to memory (and in some cases, a secondary temporary data store on disk) and used for the input of the next component in the plan. Each operation is queued upon the CPU using tasks, workers and threads, the standard access route in most operating systems (OSs), although this behaviour can be modified using secondary mechanisms such as affinity masking [41] for multi-core systems, fibre-weight threading and task prioritisation. The leftmost, or topmost (depending on implementation) component of an execution plan is typically the root noun of the query - for projections (selections) this is SELECT and at this point the result set will be rendered to the client. The method for rendering will depend upon the database driver and the client being used, but typically will be sent as a text stream which the driver will render into the appropriate structure on the client.

The process above describes the typical journey for a single query. In practice, RDBMSs can and do cope with workloads that scale to hundreds of thousands of queries per second, and there are many auxillary mechanisms in place that complement the core query optimisation process, such as in-memory caching, query plan trivialisation, plan caching and parallelism. The description as given also omits details of the transaction-based model which guarantees the success (or rollback) of a transaction and the preservation of ACID principles [42] - a protocol not unlike TCP/IP which uses a system of acknowledgements to guarantee message reception - and database locking is also omitted, for clarity.

2.4.3 The role of the cost-based query optimiser

In an RDBMS, queries are implementations of relational algebra that execute to either make changes to data from storage or retrieve data from storage. These operations are instigated by a calling application or user. Consequently, to avoid unnecessary delay in the application or to the user, database queries should be written in the most efficient manner possible. By efficient, it is meant using the least resources (this can be measured by, for example, CPU, I/O and memory grants) to return the expected result set in the fastest possible time. To enable this to happen, RDBMSs can use a variety of approaches, most common of which is cost-based optimisation [43, 44]. This involves several steps – the query is first parsed, which involves syntactic checks and checks to ensure the referenced objects exist. Next, the query is rendered into an internal canonical form compatible with relational algebra. Next, the query optimiser generates an execution plan comprising of various operations to execute the query. This process typically applies heuristic rules to the algebraic query representation to produce a series of operations in an acyclic tree – for example, one such rule might be the consolidation and simplification of multiple WHERE conditions on a single predicate.

Each operation carries a cost – this is typically a function of the CPU and I/O resource cost and this cost is a floating-point number that is relative, having relevance only when compared to the cost of other queries. Operations in the tree are executed from the leaf nodes to the root node, and so the total cost of the execution plan is the sum of all costs of all operations as measured to the root node [39]. The optimiser will attempt to produce the best possible plan by manipulating the type and order of these operations within a predefined timeout period – the plan with the lowest total cost is the one normally executed by the database engine. Therefore, generating efficient database queries with the lowest possible cost is a core consideration when tuning for performance. For this reason, tuning the queries is a logical step in dealing with performance issues, with inefficiencies in poorly-performing query structures removed or rewritten to best match the tables present within the schema.

This is appropriate not just to generate low-cost execution plans, but because structural changes to schemas can result in processes dependent on the existing structure being unable to function – for example, the amalgamation of a set of sub-tables into a single table (denormalization) with the aim of reducing joins may require changes to all applications which use queries that call data directly from the original subset of tables [45]. While this limitation to the schema definition can be overcome by the augmentation of the schema with structures like views or indexes, finding and mitigating query inefficiencies instead of schema inefficiencies can result in swifter problem resolution. Such query inefficiencies are also often easier to find, manifested by well-understood anti-patterns – to name two, queries which use cursors to iterate over data can be outperformed by set-based representations (colloquially known as ‘RBAR’ [46], or the N+1 problem [30]), and so targeting cursor- or loop-based structures is beneficial for performance; and queries which fetch more columns of data than are required for the final result set waste resources and increase query execution time (a problem known as eager fetching), addressed by limiting the columns selected in the query.

2.5 Query Optimisation

2.5.1 Overview

Query-centred performance tuning requires the queries to be accessible, which may be through storage within the application layer or definition within stored procedures, or otherwise subject to direct manipulation without significant impact to application development. The limitations of ORM tools [20, 28, 30] include but are not limited to non-parameterisation, meaning almost-identical queries can fill the RDBMS plan cache and cause unnecessary recompilations; eager fetching and the N+1 problem; the use of nested queries rather than joins, creating inefficient query execution plans; and excessively large queries which require more time to produce efficient execution plans than is available in the optimisation process. These anti-patterns have ramifications in the eventual execution plan.

Often, tuning efforts in RDBMS systems are directed away from the queries and into the underlying data structures or infrastructure. Research in this area on an implementation-specific basis are described by, among others, Chaudhuri et al. [47] for Microsoft SQL Server; Schiefer and Valentin [48] for IBM DB2; and by Dageville et al. [49] for Oracle Database. There are many mitigating actions that can be taken to counter poorly-performing queries by rearrangement of the environment in which they run or the structures that they run against, however queries themselves can often be rewritten for better performance by arranging them to return identical results but in an arrangement conducive to the creation of an efficient query plan. For example, using a set-based approach to querying a set is recognisably better for performance than a cursor-based approach due to the efficiencies of reducing the number of table- or index scans required against the underlying data sets. However, with the growing popularity and ubiquity of ORM tooling, it is often necessary to tune the RDBMS for performance despite, or because, of the presence of poorly-constructed queries, since such ORM-generated queries cannot be readily modified *in situ* in the same way that queries pre-defined in stored procedures or through in-line code can be tuned by the database administrator. These issues are expanded upon in the next chapter.

2.5.2 Normalisation and query performance

Relational databases consist of data which are arranged into columns and rows, held in tables. Tables can have inter-relationships such as adjacency or dependency (either as a parent or child) on another table [4]. These tables can be associated using (foreign) keys, where the existence of a row

in one table is dependent on the existence of a related row in another table, as defined by one or more columns.

Using this simple set of rules, a system of normalisation was initially developed [6]. Called the normal forms, there are multiple normal form levels, each arranged in an increasingly-strict hierarchy, which define under what conditions data can be split amongst tables. Third-normal form, abbreviated to 3NF, is commonly in use however there are also higher forms of normalisation; 4NF, 5NF and BNF, which further restrict the dependencies and transitive dependences allowed in the database schemata. In practice, normalisation can be a barrier to performance due to the increase in the number of JOINS required to access the data [50, 51], which in consequence increases the number of table- or index scans required to read all data from disk into memory. This is exacerbated by the different locations of that data on disk, meaning a higher proportion of non-sequential reads than would otherwise be required.

To illustrate this point, consider a database query which means ‘to display the product name and product colour of all red bicycles where the stock level was last updated after 01 Jan 2019’ from a database containing products. This can be structured in many different ways, but the performance outcomes will differ. Fig. 2.4 illustrates a query which implements this expression in SQL using a 3NF-normalised schema. Note first the complexity of the normalised query and the corresponding execution plan. Fig. 2.5 illustrates the same query, implementing the same expression, with the same result set, implemented against a denormalised schema. In Table 2.6, the relative resource consumption is compared, noting the significantly higher amounts of resources used by the first query than the second; with the additional complexity of the normalised plan realised as a full optimisation cycle rather than a trivial optimisation (implemented in Microsoft SQL Server).

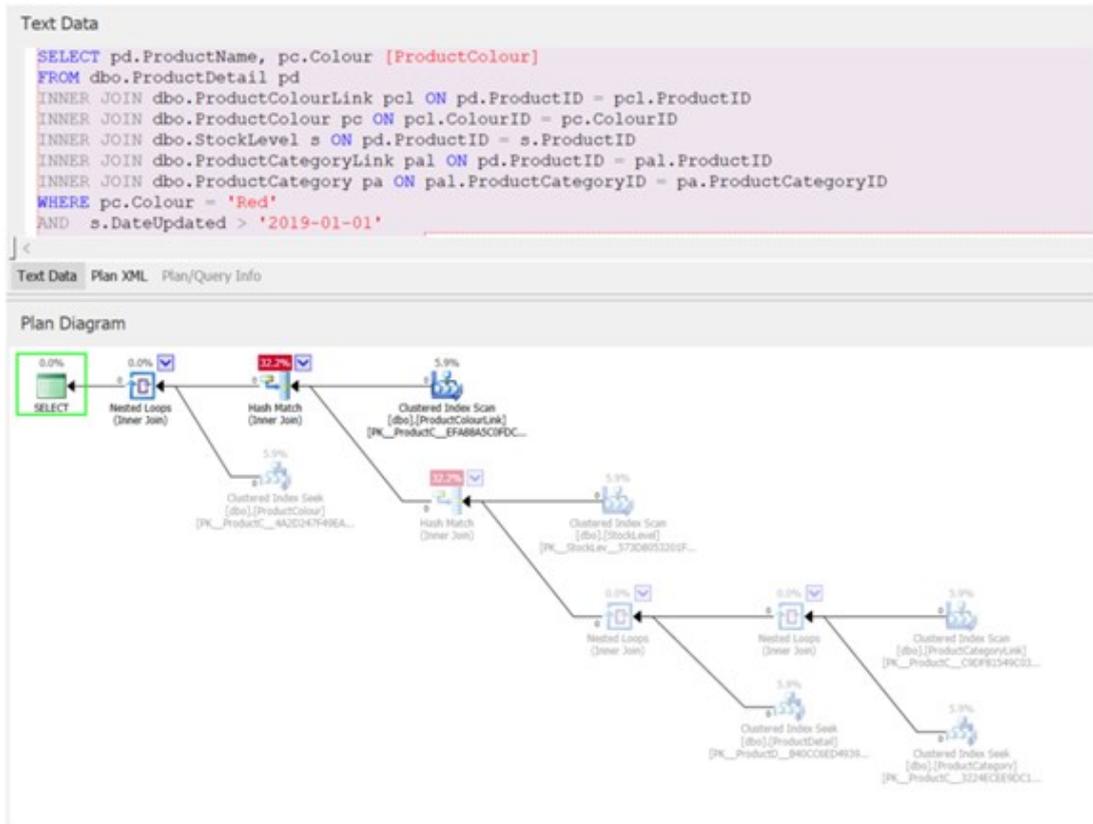


Fig. 2.4: Illustration of a normalised database query



Fig. 2.5: Illustration of a denormalised database query

Table 2.6: Relative costs compared between queries

| | <i>Normalised</i> | <i>Denormalised</i> | <i>Delta</i> |
|-----------------------------|-------------------|---------------------|--------------|
| Total subtree cost | 0.055 | 0.003 | < -95% |
| Cached plan size | 88B | 24B | -73% |
| Compiled plan memory | 752B | 136B | -82% |
| Memory grant | 2112KB | 0KB | -100% |
| Optimisation level | FULL | TRIVIAL | n/a |
| # of scans | 3 | 1 | -67% |
| # of seeks | 3 | 0 | -100% |
| # of plan components | 12 | 2 | -88% |
| # of involved tables | 6 | 1 | -88% |

2.5.3 Other query tuning strategies

One further strategy for coping with the inflexibility of database schemata is the use of views. A view is a query over some set of relations, but persisted to the database so it can be used again. Views can be categorised as materialised (or indexed), and non-materialised [52]. The former is a view which is connected to the underlying schema - if the schema changes, the view becomes invalid, since it is underpinned by an index that fetches the data referenced in the view. In this sense the view is a highly-specific index used to retrieve data corresponding to a particular query - the disadvantage of this approach is the necessity to have the index as a separate data structure from the base pages, meaning an increase in the storage required, a dependency to update this index whenever the base table is updated and additional overhead in database administration. Materialised views are non-compatible with some relational expressions, with especial difficulty encountered when implementing outer joins. Non-materialised views, on the other hand, are simply saved semantic representations of queries - when such a view is run, the view definition - the query - is run against the base tables, and consequently attracts any performance issues each time it is executed. This kind of view is created for simplicity and ease of use, but it serves only to mask the query that constitutes it.

Good schema design is intrinsic to good RDBMS performance. Recently, there is an increased focus on microservice architecture in application development [53, 54]. This paradigm is focused on the provision of small, single-purpose services that interact through common interfaces to achieve goals. Applied to database architecture, this can result in the provision of many small, single-purpose databases that hold data pertaining only to the owning application service. While this

advantage can yield benefits in code simplicity and local performance, the same schema design problems that attach to larger databases also exist in smaller ones - to normalise or denormalise, which data types are most performant for the data being stored, whether to use hard-defined foreign keys or rely on application-enforced soft keys, and how tables should relate and depend upon each other. Arguably the overhead of maintaining many small, independent databases may improve the simplicity of the system from an application development perspective, but also increases the complexity of database administration. Microservices architecture may simplify access for the application developers but there is no evidence that it alleviates query performance or schema design issues.

2.5.4 The role of schemata in query optimisation

At the time of writing, there is very little current research into better schema design in relational databases. With the improvements and recent focus on machine learning (ML) as a solution applicable to many different domains, there exists a gap in building a better schema design framework that is malleable and better-performing than single, fixed schemas, and this gap may be filled by ML-powered techniques. Chen [55] made exploratory forays into the applicability of ML to schema design, but follow-up research has been slim. This issue, and other research into this area is discussed in the next chapter, and the solution presented in later chapters describes the design and implementation of an ML-powered learning process for autonomous relational database schema design.

The existence of a pre-defined schema is integral to the concept of a relational database. As a relation is defined as a collection of related records from across one or more sets, with appropriate filters [25], *ergo* these sets must exist before a relation is formed upon them. This idea of schema-on-write, as it is also known, means that when data is recorded into a relational database the data is written into the pre-formed schema using the defined rules of SQL and the RDBMS. This differs from schema-on-read, where data is drawn (typically in non-relational database systems) from a data 'lake', or loosely-defined schema, and reformed into the appropriate configuration during runtime for the benefit of the application [56]. This latter method of fetching data is used extensively where the data itself does not form a consistent structure from record to record - data such as the content of tweets [57], or data derived from web frameworks where the attributes (columns) of the data can change in their definition from software release to software release.

Other database performance issues are rooted in schema design. Tables with large numbers of rows, for example, are subject to longer data retrieval times since the underlying heap (unindexed

table) or index can comprise many millions of pages, and index traversal times will increase accordingly. Strategies exist to counter these issues such as the use of supplementary indexes [58] and the use of in-table partitioning [59], together with infrastructural strategies such as the use of faster storage for heavily-used tables and the variation of transaction isolation levels to reduce locking [60]. Karwin [30] identifies 'god tables' as a design anti-pattern; these are tables which are so integral to the retrieval of data they are referenced more so than the average for the rest of the schema, to the detriment of performance as their associated pages are queued for access.

Denormalised databases are sometimes seen as potential solutions for the complexity introduced by normalisation and to increase the efficiency of data retrieval. Sanders and Shin [61] provide a treatment of the history of denormalisation and the performance effects on relational databases. Citing Hahnke [62], they note that denormalisation seems particularly effective in business data environments that are analytic in nature (such as data warehouses, or data marts), constructed using guidelines such as the Kimball or Inmon methodologies [63, 64]. They also note that denormalisation is a successful strategy when there is a complete understanding of application requirements available. This is a crucial point, since by design normalised databases can cope well with different application needs, and as the application matures a well-normalised schema can serve many different purposes. As Batini et al. [65] argued, the database environment should be such that "all users' data requirements and all applications' process requirements are 'best satisfied'". This aim would appear to be tangential to a database schema that is denormalised to cope with a specific application's needs, or even to the needs of a particular query. Therefore, it can be concluded that denormalisation alone is not a sufficient strategy to ensure system-wide assurance of efficient database query processing since there are trade-offs between the scalability of normalisation and the performance effects of de-normalisation.

2.6 Chapter Summary

This chapter introduced the Relational Database Management System (RDBMS) and explained the role and importance of these systems in the context of modern application software platforms, describing how traditional relational database systems face an extraordinary challenge in dealing with the growth of data generated by ever-expanding application data generation driven by societal uptake of new technologies. The rise of object-oriented programming techniques was charted, and their influence in creating object-relational impedance mismatch issues when establishing a data access layer in application architecture was examined by surveying the key literature in this area. Finally, the key steps of the query optimisation process framework within RDBMSs were discussed together with a variety of performance optimisation strategies within this framework, including

design-led approaches such as normalisation and use of views, and engine-led approaches such as query parameterisation, with reference to the literature.

Chapter 3 presents a topical literature review examining historical and current research for some of these issues in more detail, together with other selected topics closely related to the aims and objectives of this research.

Chapter 3: Literature Review

3.1 Introduction

This chapter investigates several key research areas that inform the problem definition and design of this research; database performance tuning, existing query parsing techniques, object-relational mapping technologies, information representation using graph theory and machine learning for set-theoretic applications. In line with the chosen research philosophy, these areas are investigated using a pragmatic, top-down approach rooted in grounded theory. For each area, a topical review of both historical and current research is presented, with the inclusion of other relevant material from industry where appropriate.

3.2 Literature Review Methodology

This literature review was influenced by concepts taken from grounded theory, particularly using the technique of theoretical “memoing” [1]. The literature was identified using abstract review and snowballing (following chains of previous references) as described in Chapter 1.

The following subsections give an overview of the historical and current research into the core research areas underpinning this research, extending the introduction and definition of the general themes that were presented in Chapter 2.

3.3 Database Performance Tuning

3.3.1 Overview

The importance of database performance tuning has been understood for many years. Shasha [2] simplifies the definition of database tuning as 'the activity of making a database system run faster', but tuning is also about reducing the load on the supporting systems so that concurrent transactions or other system activity are able to complete in an efficient timeframe. However, Shasha also emphasises the importance of writing database queries in such a way that they consume the least time on not just the underlying hardware, but the underlying data structures - in particular, the reduction in locking time on the data pages of the tables involved in the transaction. This paper also notes the performance implications of using a serialisable approach to sequential record inserts in B-tree structures, an observation which has stood the test of time and is still

reflected in the current advice on concurrency restrictions in serialisable transaction isolation levels as issued by RDBMS developers such as Microsoft [3].

3.3.2 The effects of data growth and maturity in query tuning

The purpose of RDBMS systems is to store and manage large amounts of structured data. Often, this data accumulates over time - indeed, records themselves might have strong temporal links, such as the accumulation and storage of log files, or for sets of financial transactions. This means that data will accumulate in the data structures and consequently, over time, the behaviour of the query optimisation process will change as the volume of the data increases. In RDBMSs, there are several approaches to managing an increasing pool of data. Horizontal partitioning [4] concerns the separation of a set of data (i.e. in a table) into several subsets based on some partition function. This is a technique supported in all major RDBMS systems and has the advantage of reducing the number of records required to be searched during a table scan, since only the partition where the record is expected to be located is targeted. Vertical partitioning, also known as sharding, is used primarily in non-relational systems since it involves splitting a table column-wise, which under the relational model would necessitate extra JOIN operations, computationally expensive.

Several attempts have been made to demonstrate the viability of vertical partitioning in relational models. Antova et al. [5] propose the extension of relational algebra with 'U-relations', a relational operation that can calculate possible rather than definite answers to a query, which inherently supports vertical partitions between tables. Cornell and Yu [6] examined vertical partitioning algorithms, citing earlier work in this field by Navathe et al. [7]. However, Cornell and Yu is limited to some extent as their approach is static in nature - the vertical partitioning is applied at the segment level (collections of physical data pages), not necessarily at the logical level; their approach applies only to lower the number of disk accesses rather than lower the complexity of the query; and as the authors state in the paper, is unsuitable for queries that access the 'non-primary' segments (the columns accessed via JOINS on the primary key column(s)) relatively often.

Rodríguez and Li [8] proposed 'dynamic vertical partitioning', a rule-based system where database queries were monitored for the attributes most commonly used, and tables vertically partitioned to accommodate the most common queries, thereby regularly changing the face of the database schemata. This latter approach is alike to the approach proposed by this research, discussed in later chapters. Today, horizontal partitioning is commonplace across all major RDBMSs but vertical partitioning has not been implemented, save within some niche features such as columnstore indexing [9] and is viewed as applicable only to NoSQL, or non-relational, database systems. The novel approach of this research project, creating divisions of the base schemata into

subsets, provides a combination of both horizontal and vertical partitioning at the tabular level to reduce the data necessary to parse to provide query responses.

Another approach to large-volume data management is archival. Considering an organisation such as a bank, it would be a reasonable assumption that the most common data accessed within the relations that store a customer's financial transactions would be the most recent ones, with less frequent accesses to slightly older data (perhaps in the form of generating statements, or aggregations of recent transactions) and almost no accesses (or none at all) to very old data, such as banking transactions from some months or years ago. In this circumstance, data archival could be a valid strategy for managing the volumes of data. Archiving data removes the rows from the tables, normally to 'cold storage' or at the least, out of the active tables. This reduces the number of rows involved in each subsequent query, speeding up accesses. There is comparatively little research available in the field of relational database archival strategies; conceivably, this could be because the movement of data is well-understood and could be seen as a common component of a business process workflow. Such an approach could be modelled as a rule-based system; for example, rows from a 'Sales' table could be archived on a regular i.e. monthly basis, removing the oldest month of data to an archival table or separate database. Such an approach is used with partitioning (see above) in so-called 'sliding window partitions', where data matching some rule is regularly re-allocated to matching partitions. Nehme and Bruno [10] present this concept as part of a wider partition management strategy in the setting of a parallel database system; an implementation of this technique in a popular RDBMS is detailed by Sundar [11].

3.3.4 Schema design and the effects of normalisation

During the design process, entities and their attributes are identified and linked, and the data flow between entities is mapped, often with tools like Entity Relationship Diagrams (ERDs). This can take place at the conceptual, logical and physical layers. Consequently, this translates organisational requirements into logical schema designs similar to application class diagrams which show how each entity interacts with others. However, in the database, the schema may be mapped differently - this is the physical schema design, and this can differ from the logical design in a number of ways - for example by normalisation, naming conventions, key management or RDBMS-specific implementation details. Martyn [12] notes that complexity in database schemas is not necessarily a problem: "If your real world is inherently complex, then your logical schemas should represent this complexity, and your users must understand this complexity in order to accurately formulate their queries". Thus, Martyn shifts the responsibility for efficient query formulation from the system to the user. However, in systems where the queries are generated by external providers

such as ORMs, this abdication of responsibility is meaningless. Instead, the schemas must themselves behave in ways conducive to good performance of the database as a whole.

To this end, normalisation (discussed in the previous chapter) is often used to model these complex relationships but has been identified as a barrier to performance. Normalisation allows the modelling of complex relationships (for example many-to-many relationships) in such a way that for all tables in schemas compliant to normal form (which range from 1NF through to 5NF, then various specialist versions such as BNF), each relation corresponds to certain rules. For third normal form (3NF), this consists of row/column intersections that contain single values; all tables have a primary key, and all non-key columns in the tables are dependent wholly and only on the primary key for the table with no transitive dependencies. This has some advantages, including the reduction of duplication in the database, but this can be at the expense of query complexity through the increase in the number of JOINS required to satisfy a query. Lee [13] recognised this as a problem of cost vs. benefit and produced a methodology for determining the extent to which normalisation should be applied to a design, using a system of decision trees, and formalised the benefits of normalisation in terms of storage space as a series of equations.

In contrast, Pinto [14] lists four principles as an argument for systematic denormalisation of previously-normalised data schemas: convenience, stability, simplicity and performance, and goes further to propose a denormalisation methodology. However, Pinto's case hinges on reducing the complexity as presented to the user and to reducing the number of JOINS. The former point is rendered invalid by the generation of queries via ORMs, leaving no human user for whom to reduce complexity, and the latter point could be mitigated through e.g. the use of materialised views on top of subsets of complex normalised schemas, or reduction in access times facilitated by faster underlying infrastructure; and furthermore, such gains may be neutralised by the increase in data volumes and therefore increased time spent on I/O operations that a denormalised schema would bring. Sanders and Shin [15] recognised these disadvantages of denormalisation and called for a balance between both normalisation and denormalisation together with a better understanding of application requirements.

Database schemas can also change over time through the introduction of new application design features which necessitate the redesign or extension of the logical database schema to accommodate the data requirements of the new features. Al-Barak and Bahsoon [16] recognised the difficulties caused by schema evolution, which they termed 'database debt', through a case study; namely, the absence of referential integrity due to restrictions in the implementation; violation of normalisation rules; violation of atomicity of values (single values in each row/column intersection), also a breach of 1NF; and overlapping tables - tables storing columns duplicated elsewhere, also called a breach of orthogonal design [17].

3.4 Query Tuning and Frameworks

3.4.1 Overview

As described in Chapter 2, database queries are implemented in SQL which obeys a common ruleset enforced by the standard [18], notwithstanding extensions to the language provided by the various RDBMS manufacturers. There are various pitfalls associated with writing database queries that can be traced, at least in part, to the influence of object-oriented thinking to a set-based, relational and functional programming environment. Karwin [19] identified various SQL 'anti-patterns' - these are patterns of behaviour that can be exhibited in both manual and ORM-driven settings. One such anti-pattern is the so-called 'N+1' problem, where rows are queried individually and repetitively before being amalgamated by the calling application. This anti-pattern exists in manual queries too but can also be enabled by the applications, and only limited assistance is provided by indexes [20].

Karwin also identified other query design anti-patterns; the so-called attribute-value pattern, where data is stored as key-value pairs in a relational database, subverts the structure of the relational model by storing attributes (columns, or domain values), in rows. Other anti-patterns are the misuse of NULLs, where blank or empty string values are substituted for NULL (or conversely, where NULL is used inappropriately). NULL has several unusual properties, including immutability and non-identity; the expression $\text{NULL} = \text{NULL}$, for example, is a contradiction. However, there are advocates for NULLs in database systems; Zaniolo [21] advanced the possibility of incorporating the concept of NULL from the implementation layer into relational algebra to represent unknown values. Another anti-pattern is to use pattern-matching inappropriately within a database query; using wildcard characters in LIKE or IN statements, for example, is very difficult to tune for since the full string of the values in any included columns will need to be parsed to determine whether they match the predicate.

3.4.2 Index-based query optimisation

Indexes can be categorised as two forms: an arrangement of data pages in such a way that the pages are accessible using a structure called a B+-tree [22], where some inherent order is required; or an arrangement of supplementary data pages that sit alongside the base table data and allow queries to access data partially or solely from this structure to satisfy a query. These structures are also in B+-tree form. Tables not in an indexed form are called heaps, which are simply collections of unordered data pages, and are the most expensive (in terms of disk accesses) structures to read from, but can be swift to write to, since pages can be written contiguously and not require page

splits or index reorganisation. This is a subject of some debate in the literature, since in the right circumstances a clustered index (index of the first type, non-supplementary) can be quicker for writes [23]. Modern database implementations use complex tree optimisation techniques to manage and access B+-trees since their initial introduction to relational databases [24]. These methods are often proprietary in commercial RDBMSs.

It is understood within the industry that a careful trade-off is required between the implementation of indexes to alleviate delays caused by excessive reads and the overhead this requires in terms of additional writes to these indexes upon table insertions, updates or deletions, the additional storage required, and the additional load on the query optimiser at run-time [23, 25, 26]. The mechanisms of indexes themselves have been the subject of much academic enquiry; Lu et al. [27] considered the use of the T-tree, an alternative to the B-tree, for memory-resident databases; Cooper et al. [28] considered the use of indexes in semi-structured data; early research noted the suitability of R-trees and their variants on non-traditional databases, including those with spatial data [29, 30]; more recently, Fuhry et al. [31] presented an indexing methodology based on B-trees suitable for use with encrypted data, and Dziedzic et al. [32] explored the possibilities of hybrid columnstore and B-tree indexes in RDBMSs. On a practical level, database administrators will look to ensure that indexes in RDBMSs are neither excessive nor missing; that they adequately cover a broad range of queries on the base tables; and that they are properly maintained, *to wit* that they are not excessively fragmented.

3.4.3 Infrastructure considerations and other mitigations

Other best practices in database management include the due consideration of the underlying infrastructure of an RDBMS. Although infrastructural considerations are not considered a primary objective of this research, some discussion is useful on the impact of the physical layer on the performance of database systems. Storage, for example, is particularly important when considering that RDBMSs will often access data pages and that these data pages ought to be as accessible and responsive as possible. Typically, then, read operations will work best across contiguous data pages (pages which are adjacent) rather than fragmented pages, the latter of which will manifest as a random I/O access pattern [33]. Traditional hard drives will fare particularly worse than solid-state drives or provisioned cloud storage due to the mechanical limitations of these drives. Therefore, it is generally accepted that RDBMSs should generally be based on servers which are solely directed towards the RDBMS and do not co-tenant with other applications or even other unconnected databases [34]. Furthermore, various best practices exist concerning the location and co-location of database files; while main database files, for example, tend to incur random reads from many different concurrent queries, transaction log files incur mostly sequential writes, and so

the two are normally separated onto different drives for better performance. Any temporary database or temporary 'scratch' files are normally located away from the main database files for similar reasons [35].

From a computational perspective, most RDBMSs support parallelism and so it is advantageous to provide multiple processor cores to service queries [36], although in rare circumstances an overabundance of processor cores coupled with the misconfiguration of the parallelism settings in the RDBMS can actually cause queries to return slower in a parallel environment than when running on a single thread, and in some cases using parallelism can result in query conditions on the applications [37]. Other environmental considerations are the amount of main memory available to an RDBMS. It is increasingly found that databases can be hosted entirely in memory; indeed, some RDBMSs support this as a feature. The advantage of doing so are vastly increased access times to the data pages, since the reliance on the underlying storage is removed; however, there is a potential for data loss using this method since in the event of power loss or other malfunction, any data not persisted could be destroyed. The operating system also has a part to play in the proper performance of an RDBMS. On Windows-based systems, for example, some configuration is required to ensure the RDBMS software has a greater degree of control over paging than other applications might require [38]

.

3.5 Existing Query Parsing Techniques

3.5.1 Context

When presented with a SQL query, the query must be transformed in such a way as to present a clear and precise algorithm to the underlying database engine. This algorithm must specify which operations to complete, in which order, and how the operation should be carried out. Although SQL queries are based upon set-theoretic concepts [39], they are also based in natural language, and this means a translation from query to algorithm is required before the query can progress through the query optimisation and execution process. In this sense there is little difference between the treatment of SQL queries to the treatment of any other higher-order language - the SQL query is compiled into a form that can be executed. This translation is called query parsing [40], or query simplification, and in essence seeks to tokenise each element of the query to identify the assets (such as data tables) and the operations (such as joins) which will then enable the computation of a viable execution plan.

The translation of a piece of original text to a taxonomy or structure against which one can compute is not a novel problem and has many overlaps in different areas of research including

natural language processing [41]. Extracting meaning from natural languages is difficult not least due to wide vocabularies, linguistic anomalies and difficulties in understanding context-based sentences [42]. However, database query parsers have several advantages over natural language-based solutions. First, the SQL language is, when compared to the full gamut of a natural language, artificially constrained in breadth. The choice of verbs is severely limited, the constructs allowed are clearly specified, and considerations such as contextual awareness are mostly non-issues. Secondly, there exists a clear set of rules for understanding the SQL language, as encapsulated in the standards, although extensions to the core SQL language are not generally platform-agnostic and implementation anomalies exist, discussed below. Thirdly, the SQL language consists of constructs which can map from the relational algebra and to a set of machine instructions, rather than the allusions and statements present in natural languages, which means constructing the path from relational expression to machine instruction is much simpler than constructing the path to an action from an extract of natural language. This precise point is also noted in Zelle and Mooney [43].

The SQL language does have a difficulty that is not present in natural language. The dialect of SQL mandated by the ANSI-SQL standards is a core set of language directives which is specified to be implemented by all relational database management systems - to put it another way, it is known and finite. However, there is no prohibition on database software implementing additional SQL language constructs above and beyond this core set of standards. This is especially true for non-relational or hybrid relational database platforms, for example the object-relational SQL standard [44, 45]. From a business perspective, this adds distinction, uniqueness and value to the product (the RDBMS) since it reduces interoperability, reduces transparency, and by increasing the complexity of doing so, reduces the incentive for consumers of these systems to migrate away in the future, thus preserving future revenue. As a result, the major RDBMS systems operate on, essentially, the core ANSI-SQL standards but implement a superset of additional features to add this unique value. In Oracle and later versions of IBM DB2, this superset is called PL-SQL, which includes the ability to interface much more closely with the underlying operating system and application programming languages [46]. In Microsoft SQL Server, this language is called Transact-SQL, or T-SQL, which extends the core language by including, for example, features like XML integration [47]. The two extensions are not interoperable or compatible.

To add confusion to the issue, sometimes even the core ANSI-SQL language specifications are construed and implemented differently: Oracle uses LIMIT to limit the results returned by a query, while Microsoft SQL Server uses TOP. IBM DB2 allows joins to user-defined functions using a special TABLE() syntax, whereas elsewhere, the syntax is to reference the function as a table directly in the join. Oracle Database allows CONTAINS(), which is unsupported by Microsoft SQL Server. Most of the RDBMSs implement the information schema (the schema containing the

metadata about the other objects in the RDBMS) differently - MySQL and its variants use INFORMATION_SCHEMA and commands like SHOW, and Microsoft SQL Server uses a combination of an information schema and dynamic management objects. Although ANSI-SQL is a widely-accepted standard, there is no effective external body, such as an enforcement agency or legal mechanism which can force adherence to these standards, and consequently the languages diverge as a result of both business considerations and software entropy, introduced over as the product lines continue to evolve.

3.5.2 Tokenisation and the parse tree

One of the first stages of parsing a database query is to identify the objects within the query and the operations upon those objects. The implementation of this varies - in Microsoft SQL Server, it is split into two stages, parsing (checking the query is valid, with the output a parse tree) and algebrisation (also known as binding), with the output an algebrised tree. The parsing stage has two functions - to check the query is syntactically valid, and to reconstruct the query in a form ready for binding to known objects and operations.

Parse trees, also known as syntax trees, are concepts that exist outside of the database domain and are applicable to many context-free grammars (such as programming languages) including SQL. However, although the language itself is context-free, some context must exist between various adjacent (or non-adjacent) terms within a database query since, for example, a join must identify two or more tables to join and the columns to join upon, each of which will be tokenised as separate elements in the tree. This relationship can be called a dependency. Pitts [48] identified how the difficulties of compiling syntax trees are compounded by this issue of binding to adjacent objects and proposed a system of higher-level classes to represent these permutations together with theorems that govern recursion and inference when constructing these trees.

In programming practice, these theorems are less abstracted. For example, Lucene syntax [49], which underpins the Elasticsearch open-source framework, includes the facility to search using tokens - that is, to break apart the terms of a query into individually-identifiable elements that can be manipulated - then use various functions such as AND, OR and tools such as thesaurus extension lookups are implemented in SOLR [50] to support the QPL (Query Programming Language) language used in some widely-known search products.

Efficient tree structures allow the accurate representation of sentences or queries for use further along the processing pipeline. Many trees of this type are dependency-based - that is to say, nodes are all terminal, and dependencies can exist between words. Covington [51] describes dependency parsing in some detail, where words in the phrase are allowed co-dependencies (analogous to

database queries) and presents a general algorithm for creating these trees. Dependency trees are suitable for finite grammars, such as SQL (where the domain of all possible words is known), as detailed by Chomsky [52] in his discussion of reduction, simplification and dependency determination - Chomsky is also a very early source for depictions of early parse trees (for natural languages). As opposed to dependency trees, there exist so-called constituent trees, where nodes may be non-terminal; in typical natural language, a non-terminal node may be a noun phrase under which exist various words in the phrase as terminal nodes, and so the noun phrase is descriptive rather than designating a specific word. However, this does not apply in the most part for database queries since these queries are dominated by individual words which have inherent meaning, rather than phrases (there is little wastage in SQL syntax), hence the use of dependency-type trees.

Covington also points out that constituency-based trees and dependency-based trees have significant overlap if the 'x-bar' linguistic restriction is placed upon the latter [53] to force all non-terminal phrases to have a single terminal node designated as its identifier, and so the difference becomes less important. Fig. 3.1 shows a simplified example of a parse tree, and an associated execution plan, for a database query. Note how the tree deals solely with the tokenisation and relationships between tokens, but the execution plan is the finished product of a binding and optimisation process that describes the operations that will take place against the database objects. By examining the components, it is shown how the execution plan is derived in part from the tree.

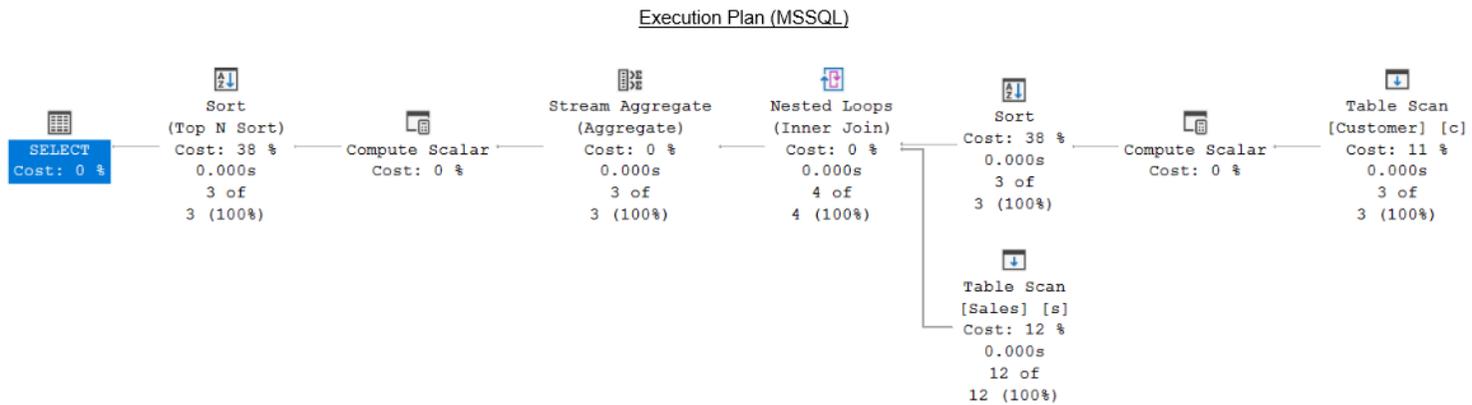
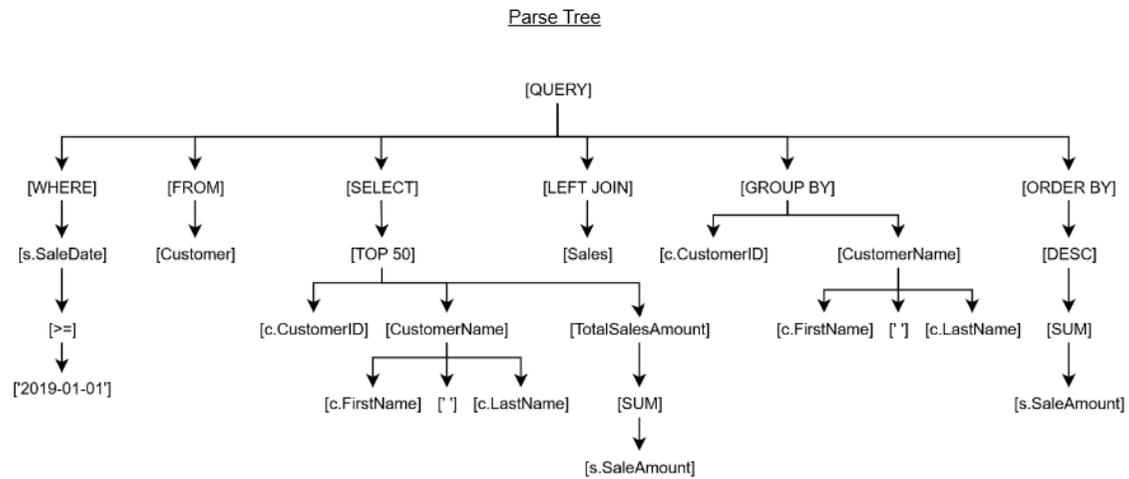


Fig. 3.1: Parse tree illustration – tokenised tree vs. execution plan

3.5.3 Query parsing in practice

The implementation of the query parser in MySQL has two elements - a lexical scanner, and a grammar rule module [54]. The former is responsible for tokenisation - deconstructing the query into atomic elements. The latter is responsible for analysing the flow of tokens (words) and identifying appropriate rules. Tokenisation is a conceptually simple technique that can be done through the application of a series of rules. For example, one rule could be to split a query into words based on some delimiter, such as a space. Fig. 3.2 illustrates the basic tokenisation of a database query. However, some other rules must come into play, as special characters are important - brackets, commas and other punctuation can alter the purpose of a statement within a query and should be included as discrete entities.

Database Query

```

SELECT TOP 50
      c.CustomerID,
      c.FirstName + ' ' + c.LastName AS CustomerName,
      SUM(s.SaleAmount) AS TotalSalesAmount
FROM   Customer c
LEFT   JOIN Sales s
ON     c.CustomerID = s.CustomerID
WHERE  s.SaleDate >= '2019-01-01'
GROUP BY
      c.CustomerID,
      c.FirstName + ' ' + c.LastName
ORDER BY SUM(s.SaleAmount) DESC
    
```

Tokenisation and Binding

| Token ID | Token | Type | Token ID | Token | Type |
|----------|------------------|-----------|----------|--------------|-----------|
| 1 | SELECT | Command | 25 | c | Alias |
| 2 | TOP | Command | 26 | CustomerID | Object |
| 3 | 50 | Literal | 27 | = | Operation |
| 4 | c | Alias | 28 | s | Alias |
| 5 | CustomerID | Object | 29 | CustomerID | Object |
| 6 | c | Alias | 30 | WHERE | Command |
| 7 | FirstName | Object | 31 | s | Alias |
| 8 | + | Operation | 32 | >= | Operation |
| 9 | ' | Literal | 33 | '2019-01-01' | Literal |
| 10 | + | Operation | 34 | GROUP BY | Command |
| 11 | c | Alias | 35 | c | Alias |
| 12 | LastName | Object | 36 | CustomerID | Object |
| 13 | CustomerName | Object | 37 | c | Alias |
| 14 | SUM | Aggregate | 38 | FirstName | Command |
| 15 | s | Alias | 39 | + | Operation |
| 16 | SaleAmount | Object | 40 | ' | Literal |
| 17 | TotalSalesAmount | Alias | 41 | + | Operation |
| 18 | FROM | Command | 42 | c | Alias |
| 19 | Customer | Object | 43 | LastName | Object |
| 20 | c | Alias | 44 | ORDER BY | Command |
| 21 | LEFT JOIN | Command | 45 | SUM | Aggregate |
| 22 | Sales | Object | 46 | s | Alias |
| 23 | s | Alias | 47 | SaleAmount | Object |
| 24 | ON | Command | 48 | DESC | Command |

Fig. 3.2: Query tokenisation example

Both PostgreSQL and MySQL uses several functions written in C to implement this tokeniser - the entry point to the tokeniser for MySQL is the `yylex()` function in the file `sql/sql_lex.cc` in the MySQL source code [55], which uses GNU Flex, an implementation of the LEX language [56]. The general process is to hash each token, look up keywords and functions against pre-existing stores, and associate symbols to each token for use by the grammar rule module. The grammar rule module takes the token stream as input and searches the stream in order to apply known rules (rules are available in `sql/sql_yacc.yy`, implemented by the Yacc compiler [57], which itself is an implementation of Backus-Naur Form (BNF) grammar notation - the compiler is implemented in C), doing this using the Bison utility [58]. Bison is a tool for converting 'annotated context-free grammar' into parse trees using a one-token LALR (look-ahead left-to-right) parsing technique. This technique was originally introduced by DeRemer [59] on the basis of the seminal paper on the LR parser by Knuth [60] on general LR parsers.

The LALR(1) parser is a simplified left-to-right, bottom-to-top parser of a token stream that does not require backtracking to apply rules and is memory-efficient. The resultant tree from the Bison output is stored in a parsing table for use by the next stage of the MySQL query optimisation process. With minor variations in the entry points for the parser and the resulting data structures, this parsing process is identical for PostgreSQL. It is not possible to assess the internal parser for some other RDBMS systems such as Oracle and Microsoft SQL Server due to the proprietary and closed nature of their source code.

3.5.4 Current research

Little evidence has been found that new query parsing techniques are being developed for use in RDBMSs; however, research into related problems using modified or existing query parsing modules is prevalent. Query parsing in RDBMSs is a subset of the wider problem of semantic or natural language parsing in information theory, and this field is active with some overlap into applicable issues for database query parsing. For illustration of this point, SPARQL [61] is a SQL-like language designed to allow queries across the so-called 'semantic web', to be used intrinsically within search engines to retrieve information based on some supplied predicates. Queries in both SPARQL and SQL use common clauses such as `SELECT` to project results based on some criteria, however some translation from natural language inputs (such as searches) is required. Better parsing methodologies compatible with SPARQL (which, by extension include SQL) have some applicability to SQL parsing [62, 63] since SPARQL is a superset of SQL, and in an older source, Zelle and Mooney [43] addressed the precise problem of mapping natural language queries to relational database queries through an experimental implementation in PROLOG.

There is some current appetite across research and industry for the modification of new query parsers to suit the purposes of the applications using query languages. This is shown by the literature - Eldawy et al. [64] propose a system for spatial data handling that includes the injection of new features for spatial data types in the Impala parser. Abstracting away from specific parser implementations, the ANTLR tool enables users to build new classes of parsers for cross-platform purposes (based on the parse tree methodology), and these have been used successfully in academic settings to build new SQL parsers both in relational and non-relational databases [65, 66, 67].

There is also at least one commercial offering available for single-component parsers, such as the 'General SQL Parser' suitable for implementation in various high-level languages [68].

In a wider context, there is significant overlap in general string parsing in NLP and clause-based query parsing, as shown in Thenmozhi and Aravindan [69] who illustrate a method for identifying paraphrases within strings (as opposed to tokenisation of individual words through delimitation, as previously described) using support vector machines. This method of grouping words could have applicability to improving the efficiency of the tokenisation process for database queries. However, NLP is arguably more concerned with the problems of analysing natural languages rather than the parsing of programming languages and as such there is little continuing current research on parsing database query languages, which are subsets of the latter.

Analysis of older research reveals some sources which shed light on how today's query parsers have been developed. Ozsoyoglu et al. [70] described a method for query parsing based on recursive pattern-matching of input database queries using a match-bind process very similar to the modern process of parsing and algebrisation, but in the context of a proposal of a summary table-by-example RDBMS. Chamberlin et al. [71] discussed the System R 'precompiler' which abstracts the parsing, binding and access path selection elements from the critical path of a transaction for the faster execution of queries, an approach used in part today in Microsoft SQL Server when using execution plan stubs for ad-hoc queries [72].

3.5.5 Query parsing limitations

Query parsers used in RDBMSs are subject to some limitations, which can be examined by an analysis of the underlying theory of the methodologies for the parsing process and examination of adjacent, related research.

The creation and evaluation of parse trees is fundamental to the query parser. Li et al. [73] question the effectiveness of tree structures for language parsing on a general basis. Although their research is in the contexts of recursive neural models and NLP, the method examined is the bottom-up generation of syntactic parse trees, an identical method to that used in parse tree

generation in MySQL and PostgreSQL RDBMSs. The authors conclude that for semantic relationship classification (the pairing or grouping of sequential or close words in some given sentence to add meaning, as required for SQL clauses such as `SELECT [column_list]` or `FROM [table_name]`), recursive modelling using neural networks can outperform standard tree creation algorithms - the time to create these is reduced. However, for discourse parsing, which also has similarities to SQL in that the input sentences tend to be short and there are relationships between sentences that need representation in the tree (in SQL, `SELECT ... FROM ... WHERE`), there were no significant differences found between the authors' new methods and the existing ones. Further work extending Li et. al to the SQL language would be beneficial in clarifying further whether any benefits are possible.

Fagin et al. [74] provide an overview of probabilistic versus rule-based approaches when discussing research into resolving ambiguities and inconsistencies in information extraction systems. Parsing of relational database queries is achieved through rule-based systems which group token streams and bind the commands and database objects to the query operations and the database objects respectively, and Fagin et. al. note that limitations of such systems are the ad-hoc nature of rule creation and the overhead of rule maintenance (for example, the parser is subject to further development as the SQL language evolves). Even if the rules are clearly defined, rule-based systems are not straightforward; Trim [75] states that tokens should be both a) linguistically significant and b) methodologically useful, and cites others [76, 77] in recognising that tokenisation is fraught with difficulties, such as recognising the differences between significant and insignificant whitespace, dealing with punctuation, and dealing with text that is improperly formatted.

These limitations open avenues for exploring query parsing alternatives, or supplementary techniques for reducing the workload sent to the query parsing process through query pre-processing. Our research project introduces a novel method for doing so using multi-dimensional adjacency matrices for query representation, and a new method of inter-query similarity scoring using statistical methods to reduce plan cache recompilations.

3.6 Object-Relational Mapping Technologies

3.6.1 Overview

Database queries are generated from a variety of sources. Increasingly, such sources include object-relational mapping (ORM) frameworks, which are interpreters between object-oriented languages (such as Java) and the set-based reality of the relational model. Using these tools, fixed SQL syntax is generated from method calls on the application side for use in the database engine, and

the database engine returns results which are translated into the appropriate application-side data structures for further use.

Object-oriented programming methods and languages have become prevalent over functional methods and languages. This has led to disparity between the class-method-interface model of object-oriented programming and the SQL query interface of the relational database; this disparity, termed object-relational impedance mismatch, has been charted in the literature [78, 79, 80] and proving the extent of this issue has been the focus of our previous research [81, 82, 83]. Ireland [78] classified this problem into four facets of a conceptual framework: paradigm, language, schema, and instance, and in response to the difficulties of overcoming the object-relational impedance mismatch problem, the industrial response was development of object-relational mapping (ORM) tooling.

In response to this mismatch, intermediary ORM software agents were developed which include the automatic generation of queries using a supplementary object-relational map, allowing developers to call a method rather than write queries directly. The language then uses this interface by calling methods, which the ORM then translates through its internal data model and into database queries, issued against the database query engine. When the result set is returned, the ORM presents the result set in the specified format. These tools have various restrictions which limit the use of conventional relational query tuning mechanisms – for example, a propensity for nesting rather than joining, row-by-row (also known as N+1) query patterns (discussed elsewhere), and eager fetching [84, 85]. These issues could be overcome with careful query tuning, but unlike traditional non-ORM queries, ORM queries are generally inaccessible for rewriting as they are generated at runtime and not stored inline, nor stored as functional code blocks like stored procedures. This can present significant difficulties when tuning for system-wide database performance since there is little control over the query execution. More generally, this use of object-oriented application development causes a clash between the object and the relational model - essentially, this is a structural incompatibility between the characteristics of an instance of an object and the data stored in a relation, such that the data in the table cannot be stored as attributes in the object on a permanent basis but must be populated via query. As objects in object-oriented programming languages can be highly variable, so too can queries.

3.6.2 Performance challenges from ORM technologies

Query performance tuning is a well-understood field in relational database management. However, relational query-centred performance tuning approaches only work when the queries are accessible for tuning, that is when they are in a format which is compatible with query tuning mechanisms such as the cost-based optimiser. Since the inception of relational database systems, the principal programming paradigm has gradually shifted to Object-Oriented Programming Languages (OOP/L)

[86, 87, 88], where objects are created and destroyed during normal application workflows and consequently database queries are generated when needed, rather than called from a query library or stored procedure.

Tuning the queries is often the first step in dealing with performance issues, with inefficiencies in poorly-performing query structures removed or rewritten to best match the tables present within the schema, as discussed in Chapter 2. This is appropriate not just to generate low-cost execution plans, but because structural changes to schemas can result in processes dependent on the existing structure being unable to function – for example, the amalgamation of a set of sub-tables into a single table (denormalization) with the aim of reducing joins may require changes to all applications which use queries that call data directly from the original subset of tables [89]. While this limitation to the schema definition can be overcome by the augmentation of the schema with structures like views or indexes, finding and mitigating query inefficiencies instead of schema inefficiencies can result in swifter problem resolution, something that can be difficult to do with ORMs since ORMs generate queries automatically based on pre-defined rules and heuristics which are not necessarily geared to produce well-tuned queries [90].

Query-centred performance tuning requires the queries to be accessible, which may be through storage within the application layer or definition within stored procedures, or otherwise subject to direct manipulation without significant impact to application development. The limitations of ORM tools include but are not limited to non-parameterisation, meaning almost-identical queries can fill the RDBMS plan cache and cause unnecessary recompilations; eager fetching and the N+1 problem [85]; the use of nested queries rather than joins, creating inefficient query execution plans; and excessively large queries which require more time to produce efficient execution plans than is available in the optimisation process. These anti-patterns have ramifications in the eventual execution plan.

ORMs are designed to mitigate many of the facets of the ORIM problem by the provision of an interface from the application layer to the data layer. Despite this, ORMs are reported to have pervasive performance issues which arise as an artefact of their design [19]. Chen et al. [80] demonstrated that these anti-patterns can include the ‘N+1’ problem; this is where a query is implemented as a series of row-by-row implementations. Although this has the benefit of being memory-efficient, from a database performance perspective this can produce an unwanted number of table or index lookups (or scans, or seeks) and can lead to an exponential overhead in query processing time and resource consumption. By the designs of relational theory, set-based queries are preferred due to better efficiency and lower query cost [19, 80, 91, 92, 93]. Chen et al. [80] also describe the eager fetching problem (‘excessive data’) where extra columnar data is brought through to the application from within the query then discarded when the results are compiled.

They demonstrated a 71% increase in performance for a set of queries when mitigating this anti-pattern.

Cheung et al. [91] repeated this finding and reported the details of how ORMs can hide this behaviour from the user, for example by using pre-fetching. The consequences of pre-fetching data include slower execution time, increased system resource use, and more data traffic. The manufacturers of ORM tools also report adverse behavioural patterns with their tools; Microsoft Corporation [94] describe 8 different performance considerations in a popular ORM tool, Entity Framework that negatively impact query performance (7 of which occur before the query is executed). They also discuss nested queries and offer commentary on the impacts of returning large data volumes on temporary data stores and overall execution time.

Karwin [19] discusses SQL anti-patterns in general but specifically identifies issues with ORM-generated queries. Models (in the Model-View-Controller arrangement) are very closely coupled with database schemata; this means changes to the schemas can result in model incompatibilities. Another related problem is inheritance; if a class is given create, update and insert capabilities, subclasses can inherit from this class which can allow direct access to the database, reducing cohesion.

3.6.3 Current research

To date, no conclusive solution to the object-relational impedance mismatch problem has been identified, and research into this area is slow. Instead, various researchers have proposed extensions and augmentations to the object-relational model to introduce new features or mitigate some of the disadvantages of using ORMs. Malysiak-Mrozek et al. [95] investigated using fuzzy logic within ORM tooling - this could provide the advantage of retrieving probable sets rather than crisp sets of data, reducing the need for re-querying, at the possible expense of further data refinement in the application. Raghu and Varma [96] propose using JSON as an alternative to an ORM layer, particularly in shared databases (databases with more than one application reading and writing from them). In industry, there are over 70 ORM frameworks available for developers [97], indicating the maturity of ORMs as a perceived solution. However, the impedance problem categorised by Ireland et al. [80] continues to exist, meaning further research into this area would be beneficial to help close the gap between the object and the relational worlds.

3.7 *Information representation using graph theory*

One focus of the research in this thesis is the presentation of a multi-faceted theoretical solution to the problem of optimising RDBMSs for the efficient processing of queries originating from non-traditional sources, such as ORM frameworks. To achieve this goal, the internal representation of queries must be considered since, as established, serious representational and optimisation deficiencies manifest during the processing of queries from these sources. At present, queries are parsed, tokenised and rearranged into trees, and the trees inform the design of the execution plan, which is translated into a series of machine-level instructions and executed. Although this model is ingrained in various modern implementations, the consideration of an alternative form of representation for SQL queries is worthwhile in establishing whether such an alternative model can reduce the costs associated with parsing a query in the tree form. To do this, an approach grounded in graph theory is detailed in Chapter 7 using multi-dimensional adjacency matrices to chart the 'shape' of a query and is used to make meaningful comparisons against other queries. This subsection of the literature review therefore introduces graph theory; and outlines and summarises historic and current research with a particular emphasis on the intersection of graph theory and information theory, particularly in terms of relational or structured information.

Many problems can be modelled using graph theory, including database relations. Consider a non-empty, simple directed graph G with $|V|$ vertices and a collection of $|E|$ ordered binary tuples representing edges, or connections between the vertices, then a new relationship between any pair of vertices can be represented by the simple insertion of an appropriate relationship, or tuple, into E . This allows for the retention of information within the graph. For example, one may model two vertices as 'Customer' and 'Purchase', which correspond to rows in the appropriate database tables Customer and Purchase. The directed edge from 'Customer' and 'Purchase' can represent the relationship 'has made a' - relationally, this may be stored as an entry in Purchase with a foreign key column for some unique Customer identifier to the primary key column of the Customer table. The direction of the edge also assists in indicating a many-to-one relationship - a customer may make many purchases, but each purchase has one and only one customer. Thus, given a relational database of customer and purchase data, one could conceivably create a bipartite graph to model the relationships between each entity, by establishing a set of Customer(s) vertices as C (one vertex for each row in the table) and a set of Purchase(s) vertices as P .

This is not an entirely new observation - Yannakakis [98] noted in 1990 that relational databases can be represented as 'directed hypergraphs', equating the vertices to the domains (columns) and the labelled edges as the rows. Other columnar information can even be encoded as properties of the edge, such as the purchase amount as an atomic numerical value included in the tuple. One can then use the properties of graph theory to derive meaningful analytics from this data; for

example, the average degree of the members of C is equivalent to the average number of purchases made by customers (as is the ratio $|C|/|P|$); $|V|$ indicates the total number of customers; and $|E|$ is the total number of purchases.

To understand how graph theory can be used to represent queries, one may look to the research of the history of using graph theory for natural language processing (NLP), since query languages are a subset of natural languages. Mihalcea and Radev [99] describe several applications for identifying key aspects of a text block, particularly keyword extraction (by the association of vertices in a graph with a ranking describing the importance of each vertex). Keyword extraction is considered as a fundamental and critical technique within NLP, since doing so assists in the categorisation of the text block within some ontology [100]. By identifying more keywords within a block, the categorisation of the block can be attained in a more fine-grained manner. Matsuo and Ishizuka [101] investigated the applications of keyword extraction including web page document retrieval, document clustering and text mining. These applications are very similar to query parsing and categorisation of database queries.

Given that vertices in a graph can contain properties, and these properties can be key-value pairs, and vertices can have edges connecting them that represent relationships, and that these edges can themselves contain properties in key-value pairs, then there is a clear parallel between relational database theory (mandating the existence of sets and relations) and graph theory, since a relation can be modelled as a graph, as described in the opening chapters of Robinson et al. [102]. This ability for graphs to contain data has led to the creation and popularity of graph databases, an alternative means of representing data to the relational model. This is distinct from the storage and processing of relational structures or queries in graph form (queries being so-called 'L-paths' in the query language L across a hypergraph [98] - a hypergraph being a graph where an edge connects not just two, but any number of vertices). Graph databases are essentially collections of key-value pairs, and relations between those pairs, stored and retrieved from an unstructured data store. This raises questions about their suitability and efficiency when compared to relational databases for storing and querying unstructured data - Vicknair et al. [103], in a study on the efficiency of relational versus graph databases for storing graph data, noted disadvantages such as the proliferation of more database objects and greater storage space (by a significant percentage), although their overall results asserted the superiority of the graph database. It is noteworthy that their comparison was on the full-text indexing capability only of the relational database rather than testing the relational model per se, and that the relational database comprehensively outperformed the graph database in all tests involving non-character data lookups.

Hypergraphs are particularly important concepts when considering the intersection between graph theory and computer science. Adjacency matrices are binary matrices in an $[X \times Y]$ form with the list of vertices along both axes whose intersections indicate whether two vertices are adjacent; that

is to say, connected by a node [104]. Conversely, incidence matrices record similar relationships but from the perspective of the edge - one axis is a list of vertices, and one a list of edges. The existence of an edge emanating from a vertex is indicated by a 1 at the intersection. Gallo et al. [105] note that hypergraphs can be modelled using incidence matrices and incidence matrices are correspondent with Boolean matrices. Given that database queries can consist of hierarchical relationships then edges could then be drawn in two ways; firstly in a graph that is not a hypergraph, by associating the column to the table independently of any association from another vertex to the column; or by using a hypergraph and having an edge from another vertex to the vertex representing the column that also passes through the vertex representing the owning table, thus connecting both the parent and child vertices with the same edge that describes the relationship with the external association. Both the former and latter methods are representable using adjacency or incidence matrices, which means that they are computable (as Boolean matrices are computable), and that this method could be used to represent the contents of any database, by extension.

There are numerous studies and surveys that seek to extend the relational paradigm into graph database theory, and vice versa. Reutter et al. [106] summarise how unions, JOINS and projections (equivalent to relational SELECTs) can be performed using a combination of types of regular-path queries (RPQs) against a graph database. They note, alongside Yannakakis [98], that such an arrangement lacks an important algebraic property - transitive closure. Transitive closure is a fundamental aspect of mathematics and relational algebra, defined as the minimal relation R on a set X that contains some defined sub-relation R' . A transitive closure can, however, be modelled in graph theory as a sub-graph G' of a graph G that contains all the directed edges of G [107]. Reutter et al. [106] assert that this property does not hold for RPQs arranged in such a manner as to enable unions, JOINS and projections, and as such weakens any purported equivalence between the relational model and graph theory, but is extendible with a special class of RPQs they introduce as 'regular queries'.

Daniel et al. [108] addresses the problem of mapping conceptual database schemas to graph databases, albeit using a non-relational implementation as an example. They note that graph databases do not have mechanisms for ensuring relational integrity, as captured in logical schema design (and enforced by physical schema implementation in RDBMSs). They use a model for mapping UML (Unified Modelling Language) to graph database principles to address these problems which incorporates a metamodel layer. This is the same conceptual design that informs ORMs and another example of the object-relational impedance mismatch problem, which implies the same kind of mapping problems [78] would result from the implementation of the meta-layer between the relational and the non-relational.

If a database query were to be modelled as a directed graph incorporating all the relational elements and operations of the query linked by relationships, then some method of similarity detection would be necessary to avoid heavy computational overhead when comparing query representations. Zheng et al. [109] identified the difficulty of executing efficient similarity searches over large graph databases, particularly how noise can influence the effectiveness of this. If representing queries, then noise could manifest as slightly different legal query arrangements, corruptions, or aliasing, or similar but non-identical queries. This precise issue of the over-production of similar but non-identical queries is a behaviour manifested by ORMs that sabotages the ability of the plan cache to operate effectively in current RDBMS systems and which could be effectively resolved by the fuzzy matching of query patterns. Fuzzy pattern matching in graph systems, and the class of this problem, fuzzy morphism, is a general problem that has a substantial current and historical presence in the literature [110, 111, 112, 113] but no universal solution.

3.8 Conclusions

Relational database performance tuning appears to have been investigated thoroughly following the inception of RDBMS systems in the early 1970s; through the 1980s and early 1990s, progress was made by a variety of seminal researchers whose names reappear frequently in both the academic and trade literature; Codd, Date, Stonebraker, Elmasri, Navathe and others. However, although RDBMSs have become entrenched as the fundamental design upon which structured data is managed and accessed by a large proportion of today's database systems, academic research from the mid- to late-1990s to today has veered towards areas in which, perhaps, more substantial improvements could be made. What has come to light during the literature review is that research progress into relational database theory itself is now almost non-existent given that the underlying principles are well-understood.

However, the context in which these RDBMSs run has changed significantly. The trend towards object-oriented programming languages became a de facto standard, so that most development today is done in languages that are based on OO principles. The repeated calls for further object-oriented features to be incorporated into RDBMSs [107] fell mostly on deaf ears - object-relational impedance mismatch meant that large barriers between the two worlds had to be overcome. The literature review has shown that performance tuning strategies that were sound for database environments running on known and finite sets of distinct queries are no longer entirely applicable to queries generated by ORM frameworks; and that providing efficient support for queries originating from these sources is a difficult and unsolved problem.

Through the review of parsing techniques, some initial exploration of query representation alternatives using graph theory were tested and found to have an academic pedigree, with related (but not identical) research into natural language parsing providing some evidence that the idea of representing queries as directed graphs may be feasible. Some parallels (and distinctions) were noted between the set-theoretic and graph-theoretic models which have translated into theoretical barriers for other researchers, and which may impede progress.

3.9 Chapter Summary

This chapter presented several areas of research relevant to the problem of improving relational database query performance in the context of increased object-relational mapping framework use, increased volume, variety and velocity of data, provided an overview of both historical and current enquiries in this sphere. The issue of query performance tuning in the context of database queries generated by non-traditional sources and several areas of research which may provide assistance in determining appropriate solutions were examined.

In the next chapter, the steps for exploring the extent of the problem through primary research using a mixed-methods approach are described and the results from this research are presented.

Chapter 4 - Problem Investigation

4.1 Introduction

It is evident from the literature that there are potential opportunities to improve cost-based database query optimisers, which still use standard lexicographic parsing techniques and which are subject to the changing tides of object-oriented mapping frameworks, increased velocity, volume and variety of data, amongst other anti-patterns. Existing techniques such as indexing, partitioning and sharding go some way towards improving performance but there is a research gap in the effective internal representation of queries at the optimiser level and potentially some value in developing the ideas of Chen [1] in creating truly dynamic schemas, beyond the limits of materialised views.

This chapter describes the methodology and outcomes of the primary research undertaken to build upon the findings of the literature review, and to investigate and verify some of the research objectives.

This chapter first describes our qualitative investigation through surveying practitioners in the field on their database maintenance and usage experiences. Thematic analysis was used to group and describe outcomes in a narrative fashion and identified areas for further investigation.

Next, the design and implementation of semi-structured interviews is described to triangulate the findings from this survey, and information was gathered on database practitioners' detailed experiences that helped validate and verify the previous findings, and which suggested several tangential directions for investigation.

Next, the initial forays are described into proving or disproving the hypothesis that queries generated automatically from ORM frameworks can be less efficient than queries generated manually by practitioners, a consequential question that has arisen from the qualitative research findings. Microsoft SQL Server 2014 is used, a relational database management platform, and a publicly-available sample database. The findings were presented from this initial set of experiments at the *IEEE International Conference on Consumer Electronics and Computer Engineering 2018* [2].

Finally, this chapter details the experiments to reproduce some of the ORM anti-patterns that our literature review uncovered and which are described indirectly by our research participants.

Experimental trials were undertaken against a real-life data set; weather buoys situated in the Pacific Ocean, which provided an interesting multivariate temporal data set upon which meaningful and lifelike database queries can be tested. The attempts to reproduce these anti-patterns and the

findings were published in the *Journal of Database Management*, together with the survey findings [3], from which some of the material in this chapter is adapted.

4.2 Domain Expert Investigation - Survey

4.2.1 Survey investigation

Given the secondary research findings on the extent of the anti-patterns exhibited by object-relational impedance mismatch (ORIM), actioned by object-relational mapping (ORM) tools, this section aims to investigate if ORIM presents practical issues, and if so the extent of these issues, by the administration of a survey focused on object-relational mapping tools, delivered to an audience of database practitioners. An investigation is mounted as to whether ORM-produced queries and ORMs in general cause performance issues in real-life database environments.

A survey was designed, piloted and delivered, consisting of 18 questions for an audience of database practitioners with the intent to investigate several topics: the proportion of respondents who use an ORM, or use or administer database systems with ORM inputs; an estimation of the proportion of query traffic to relational database systems originating from ORMs; the experiences of the respondents in working with ORM query performance tuning, schema management, big-data-fed database systems and non-relational data stores; the beliefs of the respondents in relation to the effectiveness, compatibility and integrative ability of ORM tooling; and the opinions of the respondents on ORM-related paradigms such as object-oriented programming, Big Data, the Agile software programming methodology; object-relational (hybrid) systems and automation; all tangential topics which the secondary research findings showed may contribute to the influx of data and be responsible for the object-oriented programming methodology that warrants ORM use.

4.2.2 Survey design

The questions were structured primarily using Likert scales, with a mixture of qualitative free-form textual information to gather further details without placing constraints on the responses of the participants. This approach invited respondents to express their level of agreement or disagreement with several database-specific statements on a Likert scale with an additional neutral option to allow null answers to be statistically disregarded.

Delivered via the instant-messaging platform Slack to a database-specific interest group, the survey returned 19 responses. Relational database performance tuning is very specialist area so the total

available population was expected to be small; $n = 19$ in these circumstances compromises the statistical integrity of the output analysis, but the free-form output (open-ended responses to questions) remains valuable and basic statistical analysis can be indicative of sentiments. Slack was chosen as a popular platform for specialist communities, enabling the targeting of a particular set of skilled individuals. Responses were analysed as indicative samples of opinion using qualitative analysis, with free-text commentary from the respondents treated as significant and central contributions. The methodology of thematic analysis [4, 5] is used to group the response data into categories and observations, create themes and formulate summary narratives.

Checks and balances were built into the survey design. Given that the research questions were well-defined before the survey was issued, some risk existed that confirmation bias would skew the results if the questions were put in such a way as to seek affirmation of a pre-defined perspective. To prevent this possibility, a mixture of positive and negative question forms was used when positing statements, and at several points, questions were mirrors or alternative phrasings of others already answered. This use of cross-questioning helped ensure construct and content validity and it was found to be effective during analysis of the resulting data with few contradictions in the results.

4.2.3. Survey pilot

Before deployment, the survey underwent a pilot stage after which improvements were made to the internal consistency of the survey, refinement of the topics and refinement of the terminology based on feedback from several individuals. The survey was designed to include additional free-form text fields to ensure the capture of meaningful, context-aware qualitative information to add value, hence the use of thematic analysis. This approach was successful in uncovering additional information, useful when constructing the thematic codes.

The survey questions are provided as Appendix A.

4.2.4. Analysis of results

There are several stages of thematic analysis [4], none of which are prescriptive but provide a coherent process to analysing qualitative data. The survey was designed to capture both quantitative and qualitative responses and was analysed by using all six stages of thematic analysis, from data familiarity through to thematic mapping.

The preliminary stage, in accordance with Clarke and Braun's approach [4], focuses on semantic analysis – the extraction of the key information about what is said, or written, rather than latent

analysis of the underlying meaning. The responses from the survey were analysed in this way, resulting in a preliminary codification of the data.

In the next phase, refinement of the codes and re-arrangement of the themes took place to simplify the findings. This was accomplished by de-duplicating codes, re-arranging them into a different configuration of themes, and rephrasing the codes to remove unnecessary detail. At this stage, latent analysis began to take prominence over semantic analysis. Table 4.1 shows the outcome of this phase.

Table 4.1: Final codification of the survey results

| ORM USE | NEGATIVE ORM BEHAVIOUR |
|---|--|
| Minority proportion of query traffic generated from ORMs | Parameter sniffing |
| ORMs not used across all organisations | Poor execution plans |
| Big data performance tuning not integral part of roles | Eager fetching |
| Compatible with scalable schema designs | Procedure cache misuse |
| Difficult to reproduce performance issues | N+1 row fetching |
| Challenges when designing against ORMs | Indexes not considered or supplied |
| | Lazy loading |
| EDUCATION, AWARENESS AND PERCEPTION | Nested queries |
| DBAs have fewer skills in big data administration | No contextual awareness |
| Lack of awareness in ORM internals among professionals | FUTURE OUTLOOK |
| Lack of awareness of native database tools among developers | Lack of belief in ORMs as viable future technology |
| Traditional tuning methods well understood | Automation believed to be beneficial, with caveats |
| ORMs perceived as difficult to tune effectively | Lack of belief that automation of query tuning is achievable |
| Perception that ORMs exhibit poor performance | Performance as important to viable future DB systems |
| ORM query tuning perceived as difficult | Flexibility is less important than other components |

Next, by examining the codification and theme groupings, simplifications and linkages of the concepts resulted in the interpretative creation of a thematic map. Links are drawn between concepts to show the interplay of the themes. Fig. 4.2 shows the thematic map with themes as ellipses, sub-themes as rounded rectangles, and the links and insights associated with them.

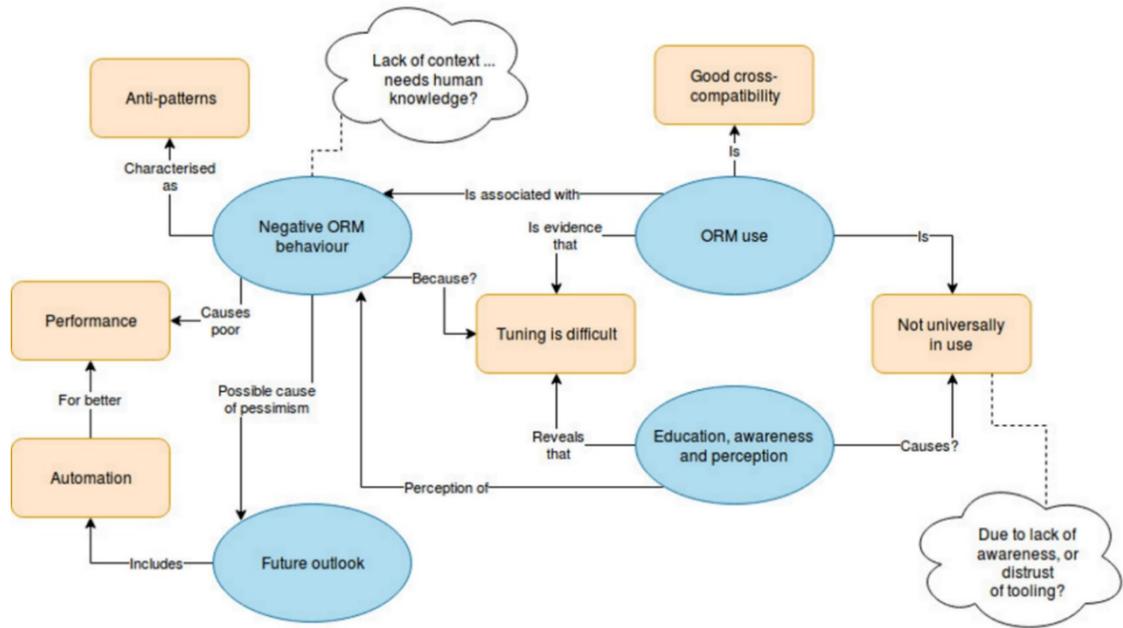


Fig. 4.2: Survey outcomes as a thematic map

The final stage was to construct narratives from the thematic map, using the notarised codes as supporting material. These narratives are presented below, and draw from the codifications, thematic map and the supporting literature.

4.2.5. Discussion of Findings

- Theme - ORM Use

The results showed that ORM uptake amongst organisations linked to respondents in the survey is approximately 60% and of those, around 25% of traffic is thought to originate from ORM tools. Consequently, ORMs are responsible for a sizable minority of query traffic. ORMs are held to be generally compatible with database scalability designs such as normalisation, but notably incompatible with some features of the RDBMS, such as re-use of plans within the procedure cache, good matching with indexes, and adherence to query structures that create efficient execution plans (such as JOINS).

The use of ORMs could be evidence that tuning databases and database queries is difficult, with the path of least resistance seen as the use of ORMs to abstract query design to an interface layer, although this finding is countered by some evidence from the comments received in the survey that there are design and interaction difficulties inherent when interfacing with ORMs, backed up with the paradigmatic differences outlined by Ireland et al. [6]. The difficulties of tuning ORMs are reinforced by a general perception amongst practitioners (67% detracting views) that this is the

case, alongside the negative consequences (anti-patterns) that arise when using them.

- Theme - Education, Awareness and Perception

There is some evidence of the view that the perceptions of ORMs as being difficult to tune are reinforced by a lack of awareness of how ORMs operate, or how they are configured, and that mutually the lack of awareness and education (of both administrative practitioners and users, or developers) contributes to the misconfiguration of ORMs – 82% of respondents had 3 or more years of experience, but only a third use ORMs regularly in their roles. There is a widespread perception that ORMs cause negative performance implications evidenced in both the free-form text responses and the statistics (no respondents agreed that ORMs were straightforward to tune), with numerous examples provided, and this could contribute to the minority use of this technology.

The responses suggest that the proliferation of ORM tools is in part consequential to a lack of awareness amongst the development community of the native tooling available within relational database management systems; for example, the use of stored procedures as interfaces, or queue-based messaging systems built into the product suite. However, this view could be biased by a cultural perception, evidenced in literature [7, 8], of a disconnection between development and administrative technical communities, manifest by the administrative audience of the survey.

- Theme - Negative ORM Behaviour

The chief finding was that query anti-patterns are held to be the causes of poor query performance in the database layer, and that this is exacerbated, with reference to the other themes, by a lack of awareness in database performance optimisation amongst developers, by lack of awareness of the native features of RDBMS systems, and by the difficulty of tuning ORM tooling. The exhibited (or perceived) behaviour of the ORM tools correlated with a generally pessimistic view of the role of ORMs in the future of database interaction, although contradicted somewhat by support for further automation. It is noteworthy that although 57% of respondents agreed automation had a role in the future of database performance tuning, only 8% (2 respondents) agreed that ORMs formed part of that role.

- Theme - Future Outlook

Automation of query- and database performance tuning was suggested both by the measured question responses and by ad-hoc suggestions in free text responses, building on prior work in the literature addressing more effective database workload management [9]. It was felt that the future of performance tuning was underpinned by automation, although emphatically not by ORMs. This

suggests that ORMs are perceived to have reached a peak performance level, and that the future of database interaction may lay in a different direction.

Several core concepts, such as performance, confidentiality, availability and flexibility were rated for importance on a scale of 1-10, with 10 as the most important. One notable result was that performance was rated at 8 out of 10, and flexibility at 6 out of 10, indicating performance to be a more important issue than flexibility, despite a flexible approach being desirable to deal with ORM-related queries.

4.3 Domain Expert Investigation - Interviews

4.3.1 Interview context

Three semi-structured interviews were carried out with chosen database professionals to collect opinions on both the current performance tuning challenges and future directions for database performance research and implementations.

In keeping with the inductive reasoning approach, these interviews were narrative, in-depth interviews conducted on loose lines of enquiry derived from the major themes that emerged from the survey (the triangulation method). Taylor et al. [10] note that this style of interview is, ‘... modelled after a conversation between equals rather than a formal question-and-answer exchange.’ This is an appropriate style where rapport is established between the participants and non-directed conversation occurs to bring out opinions and other data for later analysis.

Interview audio was recorded in full and transcribed for analysis. The method of information analysis was carried out through the extraction of opinions and ideas expressed by the interviewee using codification, with the assistance of the software package NVivo, and thematic analysis by hand, and the categorisation of these, alongside the survey output, formed a series of short conclusions and directives following an inductive narrative analysis approach. The design of the semi-structured survey, the analysis of the same and the results are discussed below.

4.3.2 Interview design

The survey output indicated four major themes that would be useful to focus upon in the interviews. Therefore, the interview questions should link to these four themes where possible. In the table below, loose question definitions are provided, inside three major categories alongside a map to one or more themes (education/training is integrated across the three major categories).

Performance tuning, as a tangential topic, is also explored (marked as [additional topic] in Table 4.3). These question structures were followed during the interviews.

Table 4.3: Mapping interview questions to survey themes

| Category | Question | Survey Themes |
|--------------------|--|--|
| ORMs | What do you know about ORM products? (define if necessary) | ORM Use |
| | What kind of query patterns etc. do you notice in systems fed from ORMs? | Negative ORM Behaviour |
| | What do you think the general perception is within the industry around ORMs? | Negative ORM Behaviour / Education, Awareness & Perception |
| | o Why do you think this is the case? | Negative ORM Behaviour / Education, Awareness & Perception |
| | Do you think that ORMs will get better over time? | ORM Use / Negative ORM Behaviour |
| | Do you think SQL is an attractive language for application developers? | ORM Use |
| | (if appropriate) What specific performance issues if any have you observed with ORM systems and SQL databases? | Negative ORM Behaviour |
| | What do you know about ORM products? (define if necessary) | ORM Use |
| | What kind of query patterns etc. do you notice in systems fed from ORMs? | ORM Use / Negative ORM Behaviour |
| Performance Tuning | In your professional practice, is database or query performance a hot topic? | Education, Awareness & Perception |
| | What kinds of tuning do you have to bear in mind (as a developer) | (additional topic) |
| | o Alternatively, what kind of database-wide tuning methods do you use (as a DBA)? | (additional topic) |
| | Has tuning become more difficult as your systems grow? | (additional topic) |
| | Are SQL databases the best solution (in your opinion) for your applications? | Education, Awareness & Perception / Future Outlook |
| | How easy do you find it to performance-tune queries that come from ORMs? | (additional topic) |
| | o Alternatively, what barriers do you find when performance-tuning ORM queries? | (additional topic) |
| Future Outlook | What impact do you think Big Data has had on managing or working with relational DBs? | Future Outlook |
| | What challenges are there around managing/working with data from the Internet of Things? | Future Outlook |
| | Do you think relational databases have a strong role to play in the future? | Future Outlook |
| | What gaps do you think nonrelational databases play in managing business data? | Future Outlook / ORM Use |
| | Do you believe the role of the database administrator is over? | Future Outlook |
| | What impact do you think cloud will have on how we manage data going forward? | Future Outlook |
| | Have relational databases reached peak performance? | Future Outlook |
| | What changes would you like to see to relational database systems to enable them to meet the challenges of the future? | Future Outlook / Education, Awareness & Perception |

4.3.3 Analysis of interview findings

The interviews were recorded with the permission of the participants. Transcription took place through an AI-augmented speech-to-text engine [11]; then the transcript was analysed by codifying specific phrases, sentiments and assertions disclosed by the participants into a range of 12 different broad topics with the assistance of the NVivo software [12]. The frequency on which these topics appeared is displayed in the graph in Fig. 4.4.

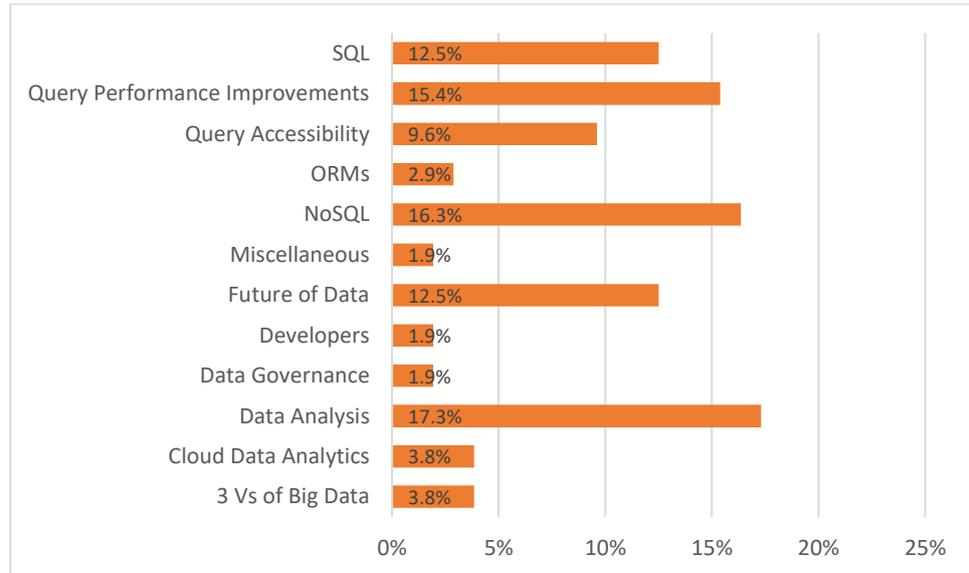


Fig. 4.4: Frequency breakdown of codified survey topics

The phrases or text segments were analysed and associated with each topic. It was determined that sentiment analysis was not appropriate since the topics are largely technical and factual in nature; the initial attempt at sentiment analysis (manually) resulted in over 50% of the codified material falling into a neutral category, and with the relatively low numbers of codified statements, the risk is run of overfitting the data. Instead, a narrative approach was adopted and for each category, some selected statements were grouped which indicate a strong sentiment for each category, and the implications of this are discussed in the following section. These groupings are presented in more detail in Appendix B.

4.3.4 *Narrative analysis*

In keeping with the pragmatic, inductive approach chosen for the research, qualitative inductive narrative analysis was selected as an appropriate technique for analysing the interview output. Although more often used in the social sciences than in the hard sciences, narrative analysis has been shown by some researchers to be applicable in the latter; this choice emulates the example of Alvarez and Urla [19] who make a strong argument for the use of narrative analysis in requirements-gathering for an ERP software installation in applying this to the technical field at hand.

The inductive approach is adopted of attempting to form an overall picture of the current state of ORM-driven SQL development and administration issues as reported by the participants with specific reference to each code above.

SQL

The participants were generally positive about SQL. Echoed in several comments were sentiments that the language is easy to learn; intuitive; universal; and can result in shorter code than object-oriented counterparts. There was some evidence that learning more complex structures and components in SQL is seen as a bar to progress.

Query Performance Improvements

Query performance was viewed and reported by the participants as being a primarily DBA-related concern; that the database should be performant irrespective of the users' commands. This was shown in an example of one particular training session. The participants reported long delays when working with big data, and organisational frustrations if data analyses were unavailable on demand. One participant reported on-premise (legacy) servers as 'feeling a lot slower'.

Query Accessibility

Participants made observations that queries are easily written for many cases; accessibility is less of an issue than it could be, but with improved training and knowledge the efficiency could be improved. Observations also included comments that writing SQL in IDE tools doesn't yield the expected range of real-time help that one would get in other languages.

ORMs

Participants had a mixed view on ORMs; some participants felt ORMs were unnecessary, that they felt more comfortable writing queries themselves; another participant reported ORMs as ‘helpful’ in this regard. Participant knowledge and experiences of ORMs was quite light with most unaware of them.

NoSQL

Reports on NoSQL and the performance frustrations in working with SQL stores was particularly evident. One participant complained that they should not be expected to remember the schemata of a structured database when moving rapidly between different data stores. Others reported that with the different types of unstructured data in use, using solely relational databases ‘doesn’t make sense’.

Future of Data

This was a topic upon which all participants had some strong sentiments. Participants expressed doubt that SQL is particularly popular and pointed out examples where other languages augmented the capabilities of SQL and, by extension, the relational model. Cloud-based systems were better able to service their needs in many cases. The view was that traditional relational database provision should be extended to include non-relational sources and capabilities; however one participant disagreed, stating, ‘...unless something really dramatically comes and takes away [relational] databases, I ... think they’re here to stay’.

Developers

There was limited data for this code, but the sentiments expressed were that developers are generally non-expert with SQL and are more used to working with APIs; consequently they needed assistance writing performant queries without access to APIs e.g. via an ORM.

Data Governance

One particularly incisive comment on data governance was an observation by a participant that their data structures were by-and-large undocumented; that there is no way of establishing data provenance, and that governance is lacking. *To wit*: ‘...I don’t want to use data I’m not going to trust, or I can’t fully, fully account for the kind of ... the lifeline where it’s come from. Because I’d *rather not have the data at all.*’ [emphasis added]. This is particularly striking and is one example of evidence for a gap in data management strategy within organisations; it may also be

symptomatic of the distrust and apathy for relational database management systems expressed elsewhere.

Data Analysis

Numerous examples were given by participants on how timely data needs to be, and how relational systems sometimes cannot meet this need. This requirement for timeliness and efficiency was evident in multiple comments and would appear to be generally indicative of the frustration in performance efficiency evidenced so far. It was noted that SQL is a comparatively easy language to learn, and that data analysts generally do not care where the data comes from providing it is accurate and timely – ‘I don’t want [the database] to take hours for me to get the data that I need’. One particularly insightful comment noted that companies are only now starting to use data in a way they have not before; that the value of data is in the use, not the collection. This could point the way to increased, targeted data collection and demand for data analysts who need efficient and performant data storage systems.

Cloud Data Analytics

In the most part the view of cloud database systems was positive. Participants noted that scalability and performance was often better; that the actual location of the data was mostly irrelevant to them; however, one dissenting opinion expressed concern over ownership and responsibility for the data if looked after by a third party.

3 Vs of Big Data

Data volume was a theme that re-occurred throughout participants’ responses. It was noted with specific examples given that large volumes of data correlated with slower performance; that larger variety of data meant analysts had to use several different systems to get the answers they needed. Commentary across various codes was indicative that volume, variety, and velocity of data is constantly increasing, and there is a need to address this.

4.4 Conclusions from Domain Expert Investigations

Triangulating the conclusions from the narrative analysis of the survey output and the thematic interview analysis, the following conclusions are reached:

- The survey showed ORM use was moderate and held to be compatible with scalability; the interview output showed mixed opinions with little data to support this view. The preference of one interview participant to write their own code corresponds with some observed anti-patterns of the ORM noted in the survey.
- The prolificacy of ORM systems to the lack of awareness in the database and developer community on the tools and techniques already available in relational database systems are linked; this correlation is borne out in the interview findings, where it was repeatedly asserted that developers tend to have basic- to intermediate-level querying abilities, and that there is a lack of focus on ensuring acceptable performance. This view is echoed in the literature [7, 8] and indicated by frequent reports of anti-patterns [ibid., and 6].
- Both the survey outputs and the interview outputs agreed that query performance is a current concern. The survey contraindicated ORMs as a potential solution, whereas the interview participants expressed little opinion on ORMs in particular; the survey participants pointed to automation as an answer whereas the emphasis from the interview participants was scalability and simplicity; both of which are achievable by automation, especially in data integration and refactoring. Survey respondents rated performance higher than flexibility, but interview participants rated schema flexibility as a key concern.

Relational database systems appear to be somewhat popular and used extensively with positive reports from the industry on their efficacy for some use-cases. However, the evidence is that with a focus on extracting value from data analysis, with an increase in the volume, variety and velocity of data collected by organisations; with the heterogeneity of data making flexibility of data schemata difficult to implement and consequently impacting relational database query performance, that there is more of a need to ensure relational database models are suitable for the changing requirements of data-driven organisations. Frustrations in the mismatch of relational systems to the query accessibility required by the developers and analysts within organisations were evident in both the primary research outputs.

The outcomes exposed highlight a need to improve flexibility and performance as key priorities within the relational database space; in the next section, some of the performance anti-patterns noted by the respondents are examined and highlighted, and attempts are made to replicate and verify the extent to which these can appear in industrial systems.

4.5 Experimental Investigations

Alongside the literature review and the primary qualitative research with industry professionals, it is sought to determine firsthand through experiment whether the ORM anti-patterns described can be replicated in current RDBMSs. These suppositions are based on the outcomes of the literature review and domain expert investigations, particularly that it is expected, based upon these outcomes, that ORM platforms will exhibit performance anti-patterns when subjected to scrutiny. This section is split into three subsections – in S.4.5.1, the investigation and outcomes from testing are presented via analysing the impact of object-relational mapping frameworks through the investigation of single queries on a sample Microsoft dataset. In S.4.5.2, more comprehensive testing of ORMs vs. traditional queries is presented by using a real-life data set (sensor data from Pacific Ocean seaborne buoys). Finally, the experimental outcomes are summarised in S.4.6.

4.5.1 Investigation of traditional queries vs. ORM frameworks

In this section, the ‘Contoso University’ Entity Framework example provided by Microsoft Corporation [13] is used against the Microsoft SQL Server 2014 RDBMS platform to seek to replicate and illustrate selected adverse effects caused by ORM-generated queries, as informed by the literature review and indicated by the qualitative primary research. In the context of a University, the following operations occur: add a student; list students; edit a student; search for a student; delete a student; and analyse the SQL generated by these processes. The application is based on the MVC design pattern with the ORM acting as the database interface. The outcomes are then examined to determine any suboptimal behaviour displayed by the ORM and, where possible, show how any performance concerns can be mitigated by a non-ORM approach, or by tuning the database or ORM.

It is important to note that for reasons of space, this demonstration is scoped to focus on some issues that emerge from single-row database calls. It is not intended, for example, to demonstrate the N+1 problem or show how ORM queries can fill the plan cache.

Adding a student

To begin, the first action is to add the student John Smith, together with their enrolment date, to the database of students using a straightforward web form. In the background, the ORM generates an INSERT statement. Notably, this statement is parameterised (the literals are passed as @0, @1 etc). An interesting point is that the student was inserted into the Person table, not the Student table, and consequently the hire date is set to NULL since it does not apply. The ‘Discriminator’ field is also interesting as it is not set by the user of the web application – when the table contents

are manually checked, it is observed that 'Discriminator' was set to 'Student'. This points to problems with the database design, but in terms of the SQL statement itself, it is correctly parameterised and does not display any performance problems.

```
INSERT [dbo].[Person]
      ([LastName], [FirstName], [HireDate], [EnrollmentDate], [Discriminator])
VALUES (@0, @1, NULL, @2, @3)
```

Getting a list of students

In this example three more students have been added to make four in total. The generated SQL is the code used to fetch this list – there are three items of data for each student, their last name, first name and enrolment date.

```
SELECT TOP (3)
      [Project1].[C1] AS [C1],
      [Project1].[ID] AS [ID],
      [Project1].[LastName] AS [LastName],
      [Project1].[FirstName] AS [FirstName],
      [Project1].[EnrollmentDate] AS [EnrollmentDate]
FROM ( SELECT [Project1].[ID] AS [ID], [Project1].[LastName] AS [LastName],
[Project1].[FirstName] AS [FirstName], [Project1].[EnrollmentDate] AS
[EnrollmentDate], [Project1].[C1] AS [C1], row_number() OVER (ORDER BY
[Project1].[LastName] ASC) AS [row_number]
      FROM ( SELECT
            [Extent1].[ID] AS [ID],
            [Extent1].[LastName] AS [LastName],
            [Extent1].[FirstName] AS [FirstName],
            [Extent1].[EnrollmentDate] AS [EnrollmentDate],
            'OXOX' AS [C1]
          FROM [dbo].[Person] AS [Extent1]
          WHERE [Extent1].[Discriminator] = N'Student'
        ) AS [Project1]
      ) AS [Project1]
WHERE [Project1].[row_number] > 0
ORDER BY [Project1].[LastName] ASC
```

Considering this data comes from the Person table, a shorter, and potentially more optimal SQL query can be assembled:

```
SELECT TOP 3 LastName, FirstName, EnrollmentDate
FROM Person
WHERE ID > 0 AND Discriminator = 'Student'
ORDER BY LastName ASC;
```

Note this differs significantly from the query generated by the ORM tool, which displays certain characteristics: unnecessary column fetches (eager fetching), namely ‘CU1’ and ‘ID’; unnecessary nesting; unnecessary sorting (the inner ORDER BY ID is overridden by the outer ORDER BY LastName); and poor alias names, decreasing the readability of the query. Even if a subquery was necessary, this could have been achieved by an explicit JOIN rather than potentially requiring the parsing of another query.

Now the execution plans of the ORM-generated query and the query proposed are compared to analyse the impact. These execution plans are shown in Figs. 4.5 and Fig. 4.6:

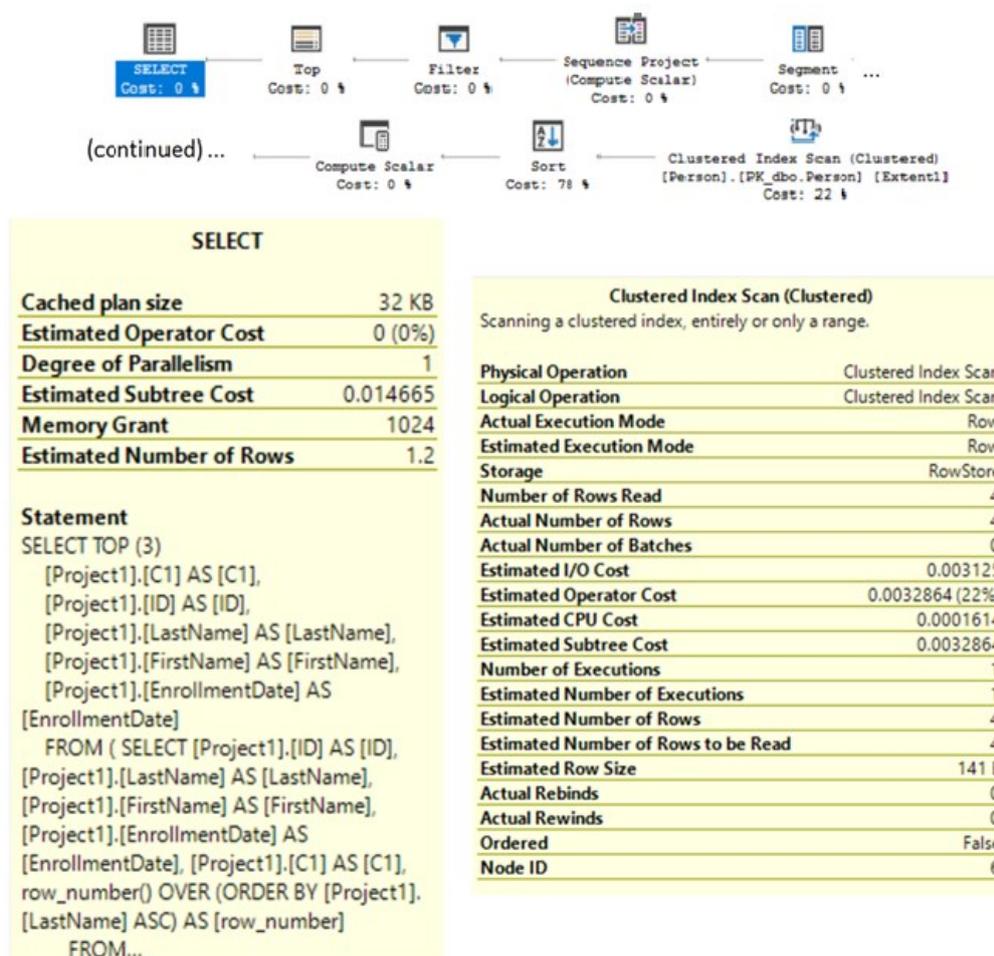


Fig. 4.5: Execution plan for the ‘list students’ query

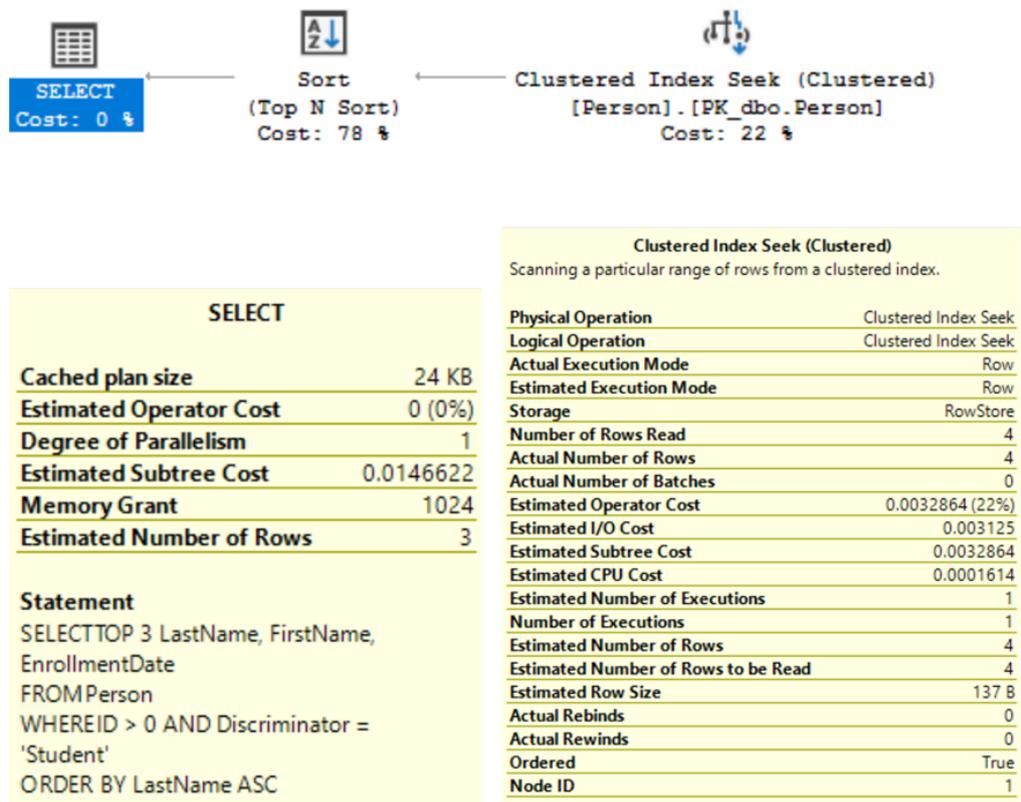


Fig 4.6: Execution plan for the non-ORM query for listing students

It is evident that the query optimiser has many more steps to execute the query. However, the optimiser has also recognised the simplicity of the queries, and this is reflected in the costs and resources used to execute them. Particularly, in addition to the comments above on the query syntax:

- The ORM query occupies 32KB in the plan cache against the proposed query at 24KB. At scale, this occupies plan cache space that could be used by other queries, negatively affecting whole-system performance.
- The ORM query underestimates the number of rows which will be returned whereas the proposed query is accurate. This can be an issue in generating well-performing execution plans at high volumes. Similarly, the estimated row size is inaccurate. Using database statistics with the ORM query could help remedy this situation.
- The ORM query uses an index scan whereas the proposed query uses an index seek. While this makes no difference in the case of low data volumes, this scales badly, with seeks on an index occupying much fewer resources than scans in every case.

Editing a student

In this example student details are edited, changing the last name and enrolment date for a single student. The output is, like the exercise in adding a student, parameterised correctly, with a simple and effective UPDATE statement. In terms of performance this is an optimal query and so needs no rewrite.

```
UPDATE [dbo].[Person]
SET [FirstName] = @0, [EnrollmentDate] = @1
WHERE ([ID] = @2)
```

Searching for a student

In this case, there are two queries executed. The first query fetches the count of results, and the second query is an adaptation of the query to fetch all students.

```
SELECT
    [GroupBy1].[A1] AS [C1]
FROM ( SELECT
        COUNT(1) AS [A1]
        FROM [dbo].[Person] AS [Extent1]
        WHERE ([Extent1].[Discriminator] = N'Student') AND (([Extent1].[LastName]
LIKE @p__linq__0 ESCAPE N'~') OR ([Extent1].[FirstName] LIKE @p__linq__1 ESCAPE
N'~'))
    ) AS [GroupBy1]

SELECT TOP (3)
    [Project1].[C1] AS [C1],
    [Project1].[ID] AS [ID],
    [Project1].[LastName] AS [LastName],
    [Project1].[FirstName] AS [FirstName],
    [Project1].[EnrollmentDate] AS [EnrollmentDate]
FROM ( SELECT [Project1].[ID] AS [ID], [Project1].[LastName] AS [LastName],
[Project1].[FirstName] AS [FirstName], [Project1].[EnrollmentDate] AS
[EnrollmentDate], [Project1].[C1] AS [C1], row_number() OVER (ORDER BY
[Project1].[LastName] ASC) AS [row_number]
    FROM ( SELECT
            [Extent1].[ID] AS [ID],
            [Extent1].[LastName] AS [LastName],
            [Extent1].[FirstName] AS [FirstName],
            [Extent1].[EnrollmentDate] AS [EnrollmentDate],
            'OXOX' AS [C1]
        FROM [dbo].[Person] AS [Extent1]
        WHERE ([Extent1].[Discriminator] = N'Student') AND
(([Extent1].[LastName] LIKE @p__linq__0 ESCAPE N'~') OR ([Extent1].[FirstName] LIKE
@p__linq__1 ESCAPE N'~'))
    ) AS [Project1]
    ) AS [Project1]
WHERE [Project1].[row_number] > 0
ORDER BY [Project1].[LastName] ASC
```

The difference between the listing and the search is an addition of another WHERE filter in the inner SELECT:

```
... AND (([Extent1].[LastName] LIKE @p__linq__0 ESCAPE N'~') OR
([Extent1].[FirstName] LIKE @p__linq__1 ESCAPE N'~'))
```

Where the test was simply to search for the string ‘Smythe’, the clause constructed uses the LIKE operator instead of the equals operator, specifically escaping the ~ sign (including it in the LIKE query). This implies the search would work for substrings also. There are also two parameters, @p__linq__0 and @p__linq__1. It is difficult to see what these parameters were, but by looking at the properties of the SELECT component of the execution plan, it is determined that they are both equal to ‘%Smythe%’.

There are three issues here: first, that both parameters were identical, increasing the complexity of the plan when one would do (there are two since they map to first and last name, but there is only one input field). Second, that the data was wrongly typed with a 4,000-character maximum. Third, that the ESCAPE clause was unnecessary since the string did not contain a tilde.

The SQL may be rewritten like so:

```
DECLARE @searchString VARCHAR(255) = 'Smythe'
SELECT LastName, FirstName, EnrollmentDate
FROM Person
WHERE Discriminator = 'Student'
AND ( LastName LIKE ('%' + @searchString + '%')
OR FirstName LIKE ('%' + @searchString + '%') )
ORDER BY LastName ASC;
```

Let us now examine the simplified execution plans (Figs. 4.7 and 4.8, below):

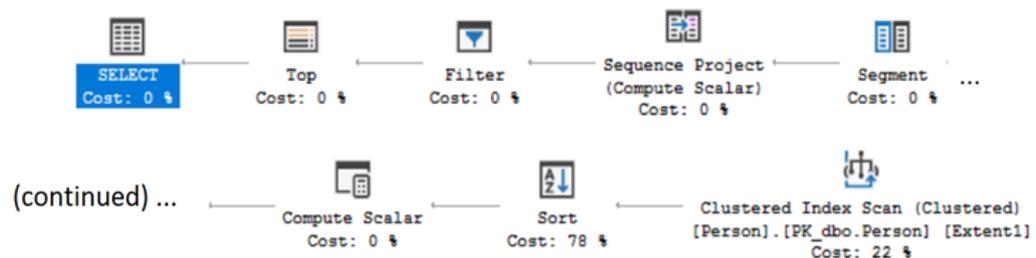


Fig 4.7: Execution plan for the ORM search query



Fig 4.8: Execution plan for the non-ORM search query

As when listing students, these plans are significantly different, and the same criticisms of the ORM query for getting a list of students apply here. However, the addition of the search predicate has altered the main operator in the proposed plan from a seek to a scan, since no appropriate supplementary index aligned with the `FirstName` or `LastName` columns exists, and the search predicate is bracketed with wildcards precluding an alphabetic scan. It is also unclear whether the Contoso search facility is deliberately designed to use wildcards or whether this behaviour is added by the ORM – if the latter, this is a worrying development since this does not reflect the original intent of the developer.

Deleting a student

As with the `UPDATE` statement when editing a student, deleting a student is also a streamlined process with an optimal query generated.

```
DELETE [dbo].[Person]
WHERE ([ID] = @0)
```

The performance can be tested, in terms of time and number of read operations, of each task that has been demonstrated. The results are shown in Table 4.9.

Table 4.9: Performance statistics for Contoso queries

| Task | Source | Logical reads | Physical reads | Parse / compile (ms) | Elapsed (ms) |
|--------|---------|---------------|----------------|----------------------|--------------|
| Add | ORM | 2 | 0 | 0 | 13 |
| | Non-ORM | - | - | - | - |
| List | ORM | 2 | 0 | 6 | 135 |
| | Non-ORM | 2 | 1 | 3 | 53 |
| Edit | ORM | 2 | 0 | 4 | 0 |
| | Non-ORM | - | - | - | - |
| Search | ORM | 1 | 4 | 4 | 290 |
| | Non-ORM | 2 | 0 | 1 | 118 |
| Delete | ORM | 19 | 0 | 7 | 19 |
| | Non-ORM | - | - | - | - |

It is noted that the performance differences are as pronounced as the differences between the query syntax structure and more so than the differences in the execution plans. From the table above, there is a significant difference in query duration between each ORM and non-ORM query pair for operations based on SELECTs – 135ms/53ms and 290ms/118ms.

Although it is recognised that the examples used here would need to be scaled to a real-world context, these results are indicative of slower performance for the ORM query. There is also a small but noteworthy difference in the time taken to parse and compile with the larger plans (ORM) taking longer to compile – 6ms/3ms and 4ms/1ms.

In summary, the observed negative behaviour of the ORM from this demonstration can be characterised as follows. Suggested mitigations in *italics* are supplied for each characteristic but note that these mitigations will in most cases require human intervention, which in turn requires an examination of the SQL generated by the ORM tool:

- Eager fetching of unnecessary columns (CU, ID)
 - o *Fetch only the columns necessary for the query*
- Unnecessary nesting (subqueries)
 - o *Avoid sub-querying unless necessary, use WHERE conditions or JOINS instead*
- Additional sorting (inner ORDER BY)
 - o *Do not use inner ORDER BY unless also using TOP / LIMIT*
- Poor code readability (particularly aliases)
 - o *Use aliases only when syntactically required*
- Poor mapping of parameters to literals (search criteria)
 - o *Use a one-to-one relationship between input parameters and SQL parameters*

- Larger execution plan size, decreasing size of plan cache for all queries (32KB/24KB, 40KB/32KB)
 - *Strive to simplify queries to lower the size of the plan*
- Unnecessary SQL constructions (nesting, ESCAPE operator)
 - *Only use the minimum structures and operators to accomplish the goal*
- Duplicate code (when fetching a count of rows and the row contents)
 - *Use functions such as ROWCOUNT or count the rows in the application*
- Apparently slower performance both during parse/compile and execution phases
 - *Simplification of the query will lead to simplification of the execution plan*

4.5.2. Investigating query anti-patterns using Pacific Ocean sensor data

The purpose of these experiments is to triangulate upon the findings of the survey, particularly around the finding that practitioners experienced performance issues when dealing with ORM-generated queries. This section investigates whether ORM tools may generate queries which have adverse performance effects when compared to queries written by a subject matter expert.

Test Data

For testing, the El Nino data set from the Pacific Marine Environmental Laboratory in Seattle, Washington, USA [14] was chosen as it contains a selection of multivariate data that lends itself to reformatting without loss of integrity and is recognised as a benchmark data set used for data mining [15], ensuring repeatability and falsifiability. This data set contains weather data readings recorded by a series of 70 buoys spread across the Atlantic Ocean between 1980 and 1998 and is presented as a single comma-separated values file with 178,080 rows and 2,136,960 data points spread across 12 attributes.

Configuration Methodology

Data were imported from a comma-separated format to a single table in Microsoft Azure DB, then normalised to 3NF to provide the advantage of simulating multi-table queries, and each column was assigned an appropriate data type. For the ORM layer, Python was configured with the Django web framework which includes the ORM tool *Django ORM*. The package *django-pyodbc-azure* was used for Azure DB database connectivity and a new model was generated from the 3NF

schema. A new property and function were created for the *distance* measurement required by one of the query objectives, detailed in the discussion of query objective O5.

Aim, Objectives and Variables

The aim of this set of tests is to examine the differences between queries generated by a subject matter expert and queries generated by an ORM tool, and note which, if any, structural anti-patterns [6, 16] are observed.

The objectives of this experiment were to determine whether:

- 1) The performance of ORM-generated queries tends to be inferior to manually-written queries when comparing execution speed, resource consumption and execution plan complexity;
- 2) ORM-generated queries demonstrate poorer relational query construction than queries constructed by a subject matter expert; specifically, whether ORMs tend to generate queries which have redundancies, are loop- rather than set-based, or display other inefficient characteristics as detailed elsewhere in the literature.

The evaluation criteria used were based upon quantifiable and measurable instruments and were chosen as accurate representations of how queries are assessed by professionals [17]. Each criterion is composed of an independent variable ('measure') whose value changes upon the manipulation of the dependent variable, and a description indicating how the criterion should be assessed ('comparative rule'). The criteria are also defined and described fully in Fritchey [17] and these are summarised in Table 4.10:

Table 4.10: Measures (independent variables) to compare the efficiency of queries

| Measure | Definition | Comparative Rule |
|------------------------------------|---|------------------------------------|
| Cached plan size (B) | The size of the cached plan in bytes. | Smallest plan |
| Total plan cost | Relative measure expressed as a real number. | Lowest plan cost |
| Compile time (ms) | Time in milliseconds to compile the plan (ready for execution). | Shortest compile time |
| Memory used during compilation (B) | Memory that was used (B) to compile the plan. | Lowest memory use |
| Memory required (KB) | Memory that was required to execute the query (KB). | Lowest memory use |
| Memory requested (KB) | Memory that the query optimiser requested to be reserved to execute the query (KB). | Most accurate (to Memory Required) |
| Total execution time | The time taken, in ms, between the query being executed and the return of the result set. | Shortest execution time |
| Total count of queries | The total number of separate SQL queries required to achieve the object. | Fewest number of queries |

The validity of objective 2, whether ORM-generated queries exhibit anti-patterns, is addressed through the comparison of each SQL query pair, noting any anti-patterns that emerge, cross-referencing against the performance analysis where appropriate and sources of query anti-patterns in the literature, and cases where query functionality is missing in the ORM.

The objectives represent queries against the data and are rendered firstly in English, then as a relational SQL query written by a practitioner; as a Django ORM method call; and as one or more relational SQL queries produced by Django ORM as a result of the method call. For brevity, these are listed in Appendix C.

The non-ORM generated queries were written to meet the query objectives before using Django ORM to generate queries that would meet those objectives. The underlying database objects via the Django ORM were accessed by opening a Django shell in Python then calling the methods in the *models* module of the new application and tracing the queries against the database using a profiling tool. This enabled the comparison of the manual database queries with the ORM queries to determine if there were any differences which might impede performance.

Experimental Results

Table 4.11 shows how the manual SQL (non-ORM) queries compare with the ORM-generated queries for the seven independent variables used as measures.

Note that due to random fluctuations in the compile time and total execution times that were outside the control of the experiment (including network latency to the database server; worker availability on the CPU scheduler; and memory allocation delays) a total of ten executions, with forced recompilation to avoid plan re-use, for each test were conducted to mitigate these effects and the mean average results (denoted as μ) are shown. Where there are multiple queries, the sum of the iterations are given under each measure heading.

Query Objective O1

The queries were non-identical. The ORM tool produced a near-identical structural query but with the addition of an explicit `CONVERT()` operation on the `airTemp` column. This conversion was not required since the column was already stored in the `FLOAT` datatype. This difference was absorbed by the query optimiser ignoring the conversion request which resulted in identical query plans.

Anti-pattern(s): *Redundant code*

Query Objective O2

Note that `aggregate()` returns a dictionary object, not a `QuerySet` object. The `annotate()` method is not suitable when there is no column to group by.

The queries were structurally identical with very small differences in the alias names and whitespace. This was reflected in the identical query plans, although the ORM-generated version took slightly longer to compile and execute, possibly due to a minute addition to the delay in the parsing stage by the different syntax.

Anti-pattern(s): *None*

Query Objective O3

The queries were structurally similar, with aliasing differences and transposition of the predicates in the `WHERE` clause. Although different query plans were used, their key metrics were identical. Of small note is how the ORM tool generated needless syntax (brackets) and did not alias the columns. Execution time was inconclusive, with the non-ORM version registering a longer execution time but the ORM version taking longer to compile.

Anti-pattern(s): *Redundant code*

Query Objective O4

Django ORM does not support the creation of Cartesian (CROSS) JOINS against the data model. Instead, a more creative solution is required. The mean average and standard deviations of the data were collected and stored as dictionary entries in memory, then the main query results similarly. The *isAnomalous* column of the main results was updated depending on the average and standard deviation values, and whether the data was missing (NULL). This overcame practical difficulties working with the *NoneType* (NULLable *QuerySet* column) when trying to convert to float. However, for a fair comparison to the SQL version, the time taken to update this *QuerySet* in memory was added. Consequently, the ORM equivalent became a three-step process.

Table 4.11. Results from ORM-generated and manual query performance testing

| | Cached plan size (KB) | | Total plan cost | | μ Compile time (ms) | | Memory used during compilation (B) | |
|-----------|-----------------------|-----|-----------------|----------|-------------------------|------|------------------------------------|-----|
| | Non-ORM | ORM | Non-ORM | ORM | Non-ORM | ORM | Non-ORM | ORM |
| O1 | 80 | 72 | 2.59791 | 2.59791 | 14.8 | 13.4 | 376 | 376 |
| O2 | 16 | 16 | 1.51565 | 1.51565 | 1.0 | 1.4 | 216 | 216 |
| O3 | 72 | 64 | 2.42003 | 2.42003 | 2.0 | 9.4 | 512 | 512 |
| O4 | 96 | 128 | 5.24644 | 5.25825 | 17.4 | 24.2 | 776 | 856 |
| O5 | 56 | 64 | 3.37822 | 13.08612 | 16.0 | 4.0 | 1344 | 472 |

| | Memory required (B) | | Memory requested (B) | | μ Total Execution Time (ms) | | Total number of queries | |
|-----------|---------------------|------|----------------------|------|---------------------------------|---------|-------------------------|-------|
| | Non-ORM | ORM | Non-ORM | ORM | Non-ORM | ORM | Non-ORM | ORM |
| O1 | 2048 | 2048 | 3152 | 3152 | 1482.0 | 1484.0 | 1 | 1 |
| O2 | 0 | 0 | 0 | 0 | 537.6 | 596.4 | 1 | 1 |
| O3 | 3240 | 3240 | 3240 | 3240 | 449.0 | 572.2 | 1 | 1 |
| O4 | 3712 | 5704 | 3712 | 5704 | 1942.4 | 1829.8 | 1 | 2 |
| O5 | 1736 | 0 | 1736 | 0 | 98.0 | 32441.0 | 1 | 1,158 |

The queries and subsequent plans produced for this query pair were extremely divergent. Due to lack of full ANSI-SQL syntax support (identified here and indirectly by Ireland et al. [6]), the approach needed to solve the problem and the consequent queries produced were correspondingly

different. The ORM-generated query also demonstrated a redundant CONVERT(), as in objective O1, but did demonstrate use of the native query preparation tools to handle the parameters and avoid storing the values with the compiled plan, which would increase the likelihood of parameter sniffing in future iterations and consequently skewed data affecting plan efficiency. As shown by the performance measurements, the ORM-generated query displayed significantly worse performance in many terms.

Anti-pattern(s): Lack of full ANSI-SQL syntax support, redundant code, multiple queries

Query Objective O5

The database query is too complex for the Django ORM to replicate directly, since it doesn't support CROSS JOIN and there is limited support for the COS, ACOS, SIN and ASIN functions. Instead, the ORM was used to extract the location data, which was consumed recursively by iterating over each row in the location data for each row in the query set, effectively recreating a CROSS JOIN. The *distances* CTE was then compiled using a custom *distances()* function in the class definition using methods from the *math* module to implement the logic. Finally, the max aggregation of the output of this function was returned to the console.

This set of calculations is an implementation of the spherical law of cosines, scaled for miles, to calculate distance between two points on a sphere [18]. This was used to accurately measure distance while taking into account the curvature of the Earth.

Observations included 1,156 individual INSERT queries ran in place of a single INSERT, the splitting up of the query into multiple queries, double writes to the database, redundant code and implicit conversion issues. Although some metrics such as plan size were smaller than non-ORM generated queries, the query execution time for the ORM query was more than 300x that of the non-ORM query.

Anti-pattern(s): Multiple queries, N+1, implicit conversion, redundant code, lack of ANSI-SQL support

Discussion

The results are assessed against the evaluation criteria as follows. For each criterion, the two results – for the ORM-generated query, and for the non-ORM generated query – are compared using the condition for the criterion specified in the ‘Comparative Rule’ column (Table 10). If the non-ORM result meets the condition, a score of -1 is assigned. If the ORM result meets the condition, a score of 1 is assigned. If the condition cannot be applied as both results are equal, 0 is assigned.

For illustration: For the criterion *Total Execution Time (ms)*, the comparative rule is *Shortest execution time*. The results obtained for this criterion across the 5 query objectives were as follows, in the format Non-ORM/ORM: 1482.0/1484.0, 537.6/596.4, 449.0/572.2, 1942.4/1829.8 and 98.0/32441.0. So, for each pair, the smallest value is found, and the appropriate score assigned. Comparing each pair, the scores are assigned as described: -1, -1, -1, 1, -1. Summing these scores yields -3. Consequently, the score for this criterion across all query objectives is -3.

In Fig. 4.12, the scores for all 7 criteria are presented using this scoring mechanism. Negative scores are associated with non-ORM generated queries, positive scores with ORM-generated queries and zero scores with neither category. For every evaluation criterion, the results showed that non-ORM generated queries outperformed ORM-generated queries using the definitions of the respective comparative rules.

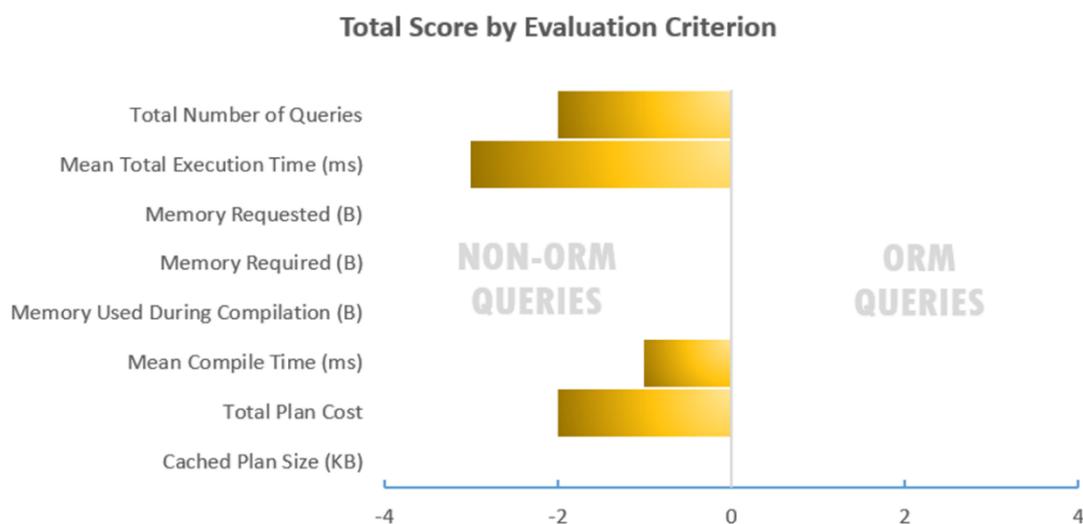


Fig. 4.12: Total Score by Evaluation Criterion

The data can also be presented pivoted by query objective (O1 to O5). For this analysis, the same scoring mechanism is used but instead of assessment solely by evaluation criterion, the assessment is by query objective, which helps illustrate the relationship between query complexity and superiority of method. The comparative rules of the evaluation criteria are used to assign scores, as before.

For illustration: For query objective O4, each pair of results is assessed against the respective comparative rules. The results are in the format Non-ORM/ORM: 96/128, 5.24644/5.25825, 17.4/24.2, 776/856, 3712/5704, 3712/5704, 1942.4/1829.8 and 1/2. The comparative rules for each can be summarised as ‘find the smallest value’, and so for each pair the smallest value is found, and

the appropriate score assigned: -1, -1, -1, -1, -1, -1, 1, -1, which sums to -6. Therefore, the score for Objective O4 across all criteria is -6.

The scores for the query objectives across all evaluation criteria are illustrated in Fig. 4.13. There is a correlation between query complexity and score – query objective O1, a simple query, had better overall performance when generated by an ORM than otherwise. Query objective O4, a complex query, had significantly better performance when generated by a non-ORM method than by the ORM, with only one evaluation criteria rating the ORM as better-performing.

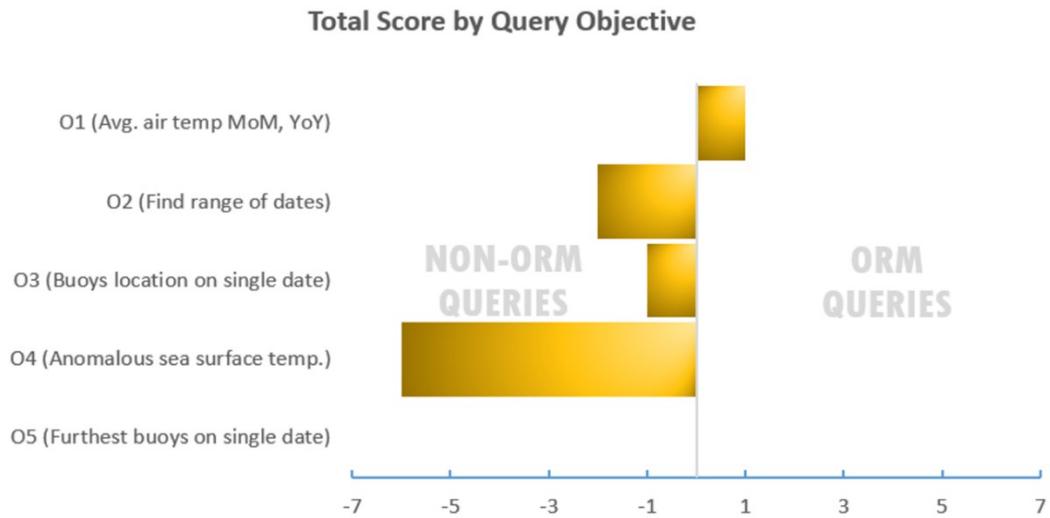


Fig. 4.13. Total Score by Query Objective

However, query objective O5 shows a neutral result despite the complexity of the query, and the reason is that the number of evaluation criteria that favoured non-ORM generated queries was equal to the number favouring ORM-generated queries, so no clear determination can be made. This highlights a weakness in this analysis approach –each criterion is given equal weighting in the scoring despite extremes in the data and comparative importance of each criterion. Query execution time can be thought of as a strong desirable trait in query performance outcomes, perhaps more so (from the user’s perspective) than plan cost or memory use, and an equal weighting for all criteria obfuscates this view. This weakness can be overcome by drawing upon the data in detail. Fig. 4.14 illustrates the relationship, drawn from the results between mean total execution time and query objectives, or complexity (where O1 is least complex and O5 most complex).

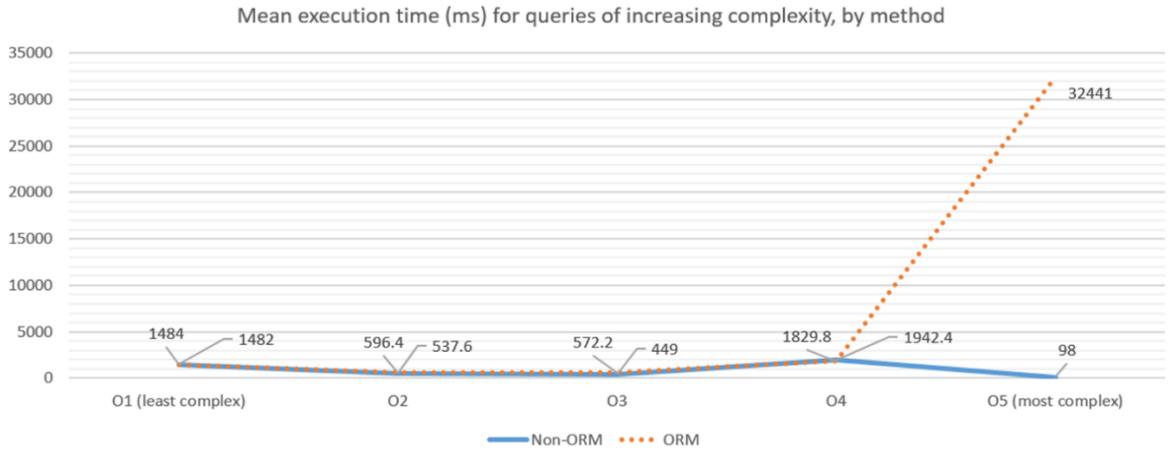


Fig. 4.14: Correlation between increasing complexity and execution time of ORM methods

This result shows that there is a generally positive correlation between complexity of query and the time taken to execute the query derived by the ORM-generated method, even if the result from O5 as an extreme outlier is excluded.

Performing t -testing on the observations of the mean execution time across the query objectives, this analysis is borne out by the p -values obtained for all the observations. Table 4.15 shows that in these t -tests, $p > 0.05$ (highlighted). This does not support the conclusion that there is a significant uplift in the execution time of the ORM-generated queries than the non-ORM generated queries as complexity rises, although other statistical measures such as mean average do support this case.

Table 4.15: p -values from t -testing of mean execution time observations

| | <i>Non-ORM</i> | <i>ORM</i> |
|------------------------------|----------------|-------------|
| Mean | 901.8 | 7384.68 |
| Variance | 600811.08 | 196496129.3 |
| Observations | 5 | 5 |
| Hypothesized Mean Difference | 0 | |
| P(T<=t) one-tail | 0.180 | |
| P(T<=t) two-tail | 0.360 | |

In general, the results showed that as query complexity rose, ORM-generated queries incurred performance penalties across multiple evaluation criteria and started to exhibit performance anti-patterns referenced in the literature [1, 16].

The scope of the investigation was over a relatively small data set of three tables. The results, showing divergence across many of the performance measures between both ORM-generated queries and non-ORM generated queries, are likely to diverge further as the complexity of the database schema and the amount of data involved increases, a conclusion supported by the evidence from the survey detailing performance deficits in ORM tools from database practitioners.

4.5.3 Conclusions from the experimental investigations

It is concluded that the evidence of query performance anti-patterns arising from ORM-generated queries is tangible, and that this phenomenon occurs for a variety of reasons. It was determined that there is a correspondence between the effects described in the technical literature and real-life measurable effects on query performance, particularly as queries grow more complex, although it is noted that ORMs can produce simplistic database queries that perform on a par with traditional database queries, and that current mitigation strategies such as parameterisation are largely unaffected, except where unused within the ORM configuration. It was found that, at scale, such effects characterise a slowdown in overall performance as the impacts of slower individual query executions cause cumulative performance effects. This was demonstrated both through individual example, and example *en masse*, and that the findings in this area have been peer-reviewed.

4.6 Chapter Summary

In this chapter, the survey instruments used to investigate current perceived weaknesses in relational database query approaches and other associated topics were detailed; this included the piloting and administration of a questionnaire to an audience of practitioners, the results from which were analysed thematically and combined with the output of a short series of expert interviews. Strong evidence was found that relational database performance and schema flexibility are ongoing current concerns, given the increase in the velocity, volume and variety of data; both in size and types; and that there is doubt among practitioners on the ability of ORMs to provide the scalability and robustness required to address these needs. It was found the move to NoSQL (non-relational) database systems to be driven by, at least in part, the perceived inflexibility of relational queries and concerns over the timeliness of results generated by data analysts.

It was further examined exactly what undesirable effects can be replicated within relational database systems by conducting two experiments; the first, to examine whether queries written manually tend to outperform ORM-generated queries, and the second, to determine whether query anti-patterns mentioned by the participants and elucidated in the literature can be reliably reproduced against a real-life data set. Strong evidence was found that in some cases, ORMs fall into some evident anti-patterns, particularly the N+1 problem, nested queries and poor cached plan re-use due to excessive recompilations. It is noted that these issues have a direct negative impact upon performance, correlating to the sentiments of the survey participants.

In the next chapter, the proposed solution is detailed, incorporating three major components; first, a new internal query representation to replace semantic text parsing and plan cache storage with queries instead stored in multidimensional matrices, computable and comparable; second, a new comparison mechanism for said matrices, using the k-nearest neighbour algorithm; and third, a new proposed method for set representation in the relational model using dynamic schema redefinition with roots in the Zermelo-Fraenkel axiomatic schema of separation. This framework is dubbed PETAS: Performance Tuning for Adaptive Schemas.

Chapter 5 - Solution Design

5.1 Introduction

This chapter introduces Performance Tuning with Adaptive Schemas (PETAS), the proposed design approach for solving some of the problems illustrated in previous chapters in the field of database query performance tuning. Specifically, this chapter seeks to address improving ORM query performance through introduction of a new method to group and compare queries by set construction, rather than as narrative text; through using this new representation to introduce better query comparison techniques, reducing recompilations on the query plan cache, a notable feature of ORM-generated queries; and through the introduction of a dynamic schema redefinition method to reduce the number of accesses required to service data queries and enable the flexibility and scalability demands of database users.

5.2 Context

Since the inception of relational database systems, the principal programming paradigm has gradually shifted to Object-Oriented Programming Languages (OOPL) [1, 2, 3], where objects are created and destroyed during normal application workflows and consequently database queries are generated when needed, rather than called from a query library or stored procedure. This use of object-oriented application development caused a clash between the object and the relational model, a problem known as object-relational impedance mismatch [4, 5]. Essentially, this is a structural incompatibility between the characteristics of an instance of an object and the data stored in a relation, such that the data in the table cannot be stored as attributes in the object on a permanent basis but must be populated via query.

In response to this mismatch, intermediary ORM software agents were developed which include the automatic generation of queries using a supplementary object-relational map, allowing developers to call a method rather than write queries directly. These tools have various restrictions which limit the use of conventional relational query tuning mechanisms – for example, a propensity for nesting rather than joining, row-by-row (also known as N+1) query patterns, and eager fetching [6, 7]. These issues could be overcome with careful query tuning, but unlike traditional non-ORM queries, ORM queries are generally inaccessible for rewriting as they are generated at runtime and not stored inline, nor stored as functional code blocks like stored procedures. This can present significant difficulties when tuning for system-wide database performance since there is little control over the query execution.

Fig. 5.1, derived from Delaney [8], illustrates the typical query execution lifecycle in an RDBMS.

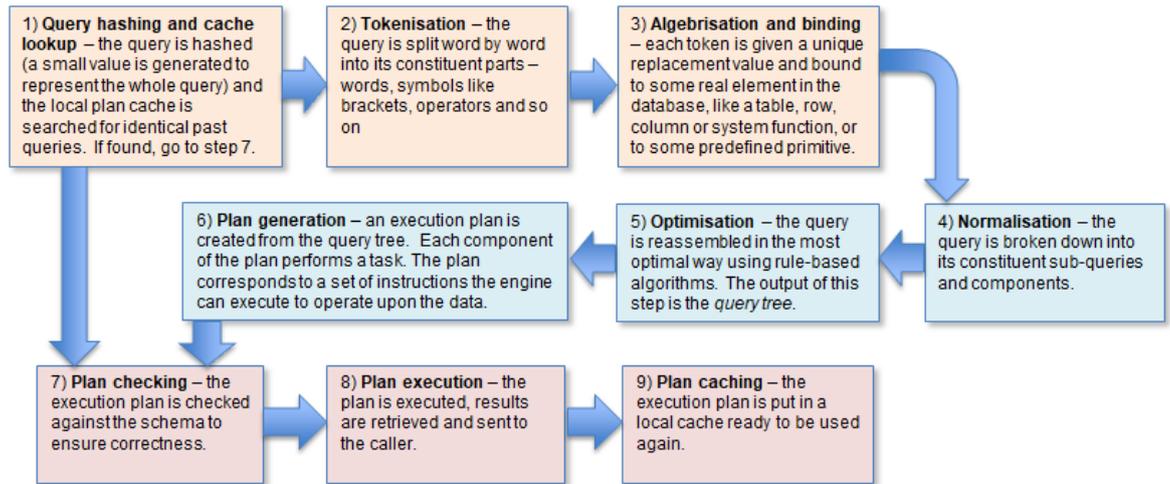


Fig.5.1: The query execution lifecycle (derived from Delaney [8])

In the proposed solution, the goal is to improve elements 1), 2), 3), 8) and 9) from Fig. 5.1.

5.3 Solution Overview

This section introduces PETAS, Performance Tuning with Adaptive Schemas, the proposed solution to the problem of tuning queries that are not accessible to relational query tuning mechanisms. PETAS is designed to complement existing strategies such as index maintenance, data archival, tuning system parameters, and application of best practices in storage architecture. This chapter describes the theoretical underpinnings, the components of PETAS and discuss the construction and results from testing a proof-of-concept implementation in PostgreSQL.

PETAS is based on the concept of multiple logical representations of data mapped to the physical data. The idea of multiple logical representations of data is not new, and already exists at the object level - indexes, views and partitions of data augment physical data structures and support query performance [9, 10, 11]. However, no such implementations exist at the whole-database level, although some theoretical work has been done on using multiple schemas [14] and schema integration [15, 16]. The use of multiple logical representations of data allows dynamic redefinition of the structure of the data to suit the inbound query flow, with the ongoing creation and destruction of secondary schemas depending on how well they perform against the context of a constantly-variable query flow. This allows the physical data pages to have more than one direct mapping to a logical schema object. Querying these different schemas with functionally equivalent (albeit syntactically different) queries would result in the same data being returned.

In current RDBMS implementations, tables are mapped to data pages either through primary indexes, which are B+ tree representations of the data together with pointers arranged in a tree or stored in heaps – unstructured collections of data pages. Both can be overlaid with secondary indexes consisting of an array of pointers of which there are various types. However, indexes of any type are only applied to individual objects – tables or materialised views. The multi-schema approach would consist of schemas containing only whole-database indexes, with the data stored once in a base schema in the normal B+ tree format and alternative schemas constructed of whole-schema index representations incorporating not only the individual objects but their relations, keys and other supplementary structures.

The use of multiple schemas is supported by a simple machine learning classification algorithm, K-nearest neighbour. The purpose of the classifier is to map each inbound query to an individual schema at runtime, depending on the structure of the query and to assess the prior performance of similar queries against the secondary schemas. This feedback is used to improve the accuracy rate of the classifier over time by monitoring and learning from the performance metrics of the query flow. In the proof-of-concept, it is demonstrated that this approach is not only viable, but that queries can be classified to different schemas effectively, and that query classification results in real and measurable performance improvements in query execution time when compared against the same queries run against a single base schema. Results are also demonstrated that indicate that the query-to-schema classifier improves in accuracy over time by learning which schemas are best suited for different schema types, through the constant self-refinement of the classifier’s own metadata.

There is little prior work in the literature on either multiple-schema models or the integration of artificially-intelligent methods for relational query performance tuning, and with effective data management increasingly important in a 'big data' culture [17, 18] and the continuance of the object-relational impedance mismatch challenge, the time is right for new research into how relational database tuning methods can be further developed.

5.4 Principal PETAS components

PETAS is a process split into two sections, the synchronous section which executes during the runtime of a query, and the asynchronous section which conducts operations outside the query execution process. The synchronous section is comprised of 5 ordered steps – the matrix parser, the scoring mechanism, the KNN selector, the schema classifier and the query mapper, the output of which feeds into the normal relational query execution cycle. The asynchronous section comprises of the metadata update process and the schema mutator. All these components are related and their position in the normal course of query execution is illustrated in Fig. 5.2:

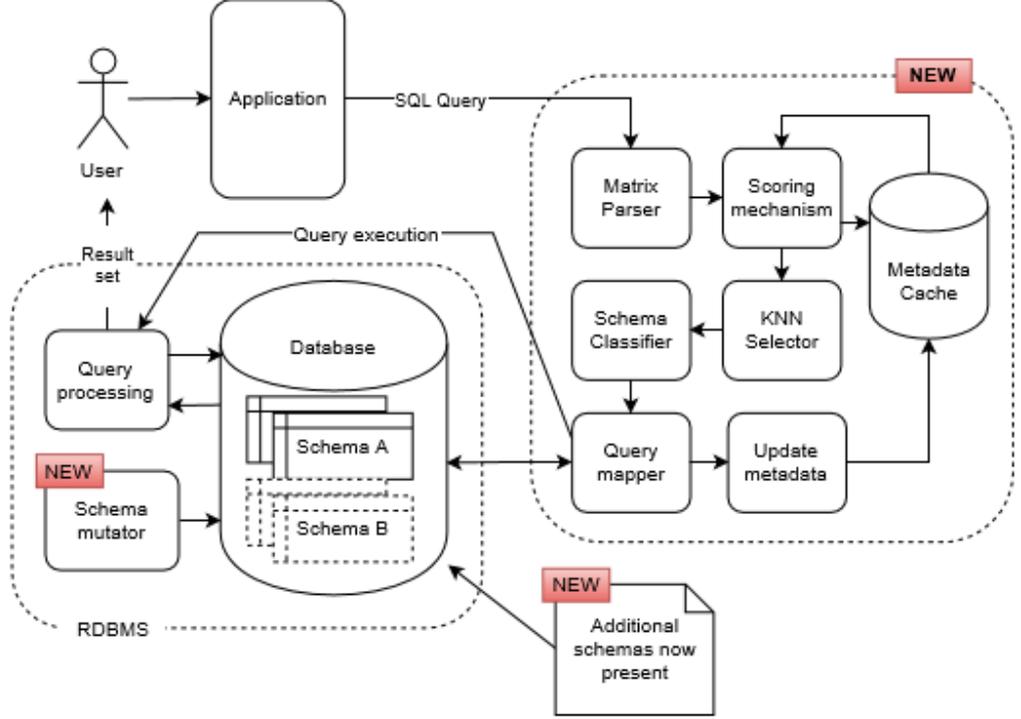


Fig. 5.2: Overview of PETAS

PETAS starts at the point that a query, Q , is received via the application from a user. Q is a valid database query against the base schema, S_b . The overall goal is to find the schema S_n (from a range of available schemas including the base schema S_b) for Q so that Q executes in the least time and consumes the least resources in comparison to all other available schemas. In the first step, PETAS deconstructs the query into a structure called a multi-dimensional adjacency matrix, which is a binary matrix in three dimensions describing connections between the components (columns, relations, parameters) within the query. In the second step, PETAS compares this matrix against the matrices of previously-executed queries and attempts to isolate K number of previously-run queries ($Q_1 \dots Q_k$) structurally-similar to Q from the PETAS metadata cache using the K-nearest neighbour (KNN) algorithm. In the third step, PETAS looks up the previous schema assignments of these ‘neighbours’ (queries $Q_1 \dots Q_k$) and fetches the majority verdict. This verdict is determined by asking, of all ($Q_1 \dots Q_k$), which schema choice in ($Q_1 S \dots Q_k S$) occurred most often? This schema is S_n . In the fourth step, as Q is only syntactically valid against S_b , Q must be mapped to a new query Q' , which is a representation of Q that is syntactically valid against S_n . Finally, in step five, the query Q' is sent to the normal query process (parser, algebraiser, optimiser and executor) supplied within the RDBMS.

The principal components of the synchronous operations of PETAS are the matrix parser, responsible for representing the query in the form of a multi-dimensional adjacency matrix; the

scoring mechanism, responsible for comparing two queries and producing a measure of similarity; the KNN selector, responsible for choosing the K -nearest neighbouring queries to the query being assessed (nearest-neighbour can be construed as ‘closest in structure’); the schema classifier, which uses the outcome of the KNN selector to assign the query to a schema; and the query mapper, responsible for syntactic redefinition of the query to match the chosen schema.

Other asynchronous functions feature alongside this critical path. First, query metadata is stored in a metadata cache. This cache is used only by the PETAS process and stores performance metadata, query weightings and definitions. The update process is asynchronous so as not to interfere with query processing. The other asynchronous process is the schema mutator, responsible for both the creation of new schemas through query assessment, and destruction of under-used schemas.

The alternative query representation and similarity scoring processes are structured together into six distinct steps, which are shown in Fig. 5.3. Dynamic schema redefinition is dealt with separately as it is both disparate and asynchronous to real-time query processing.

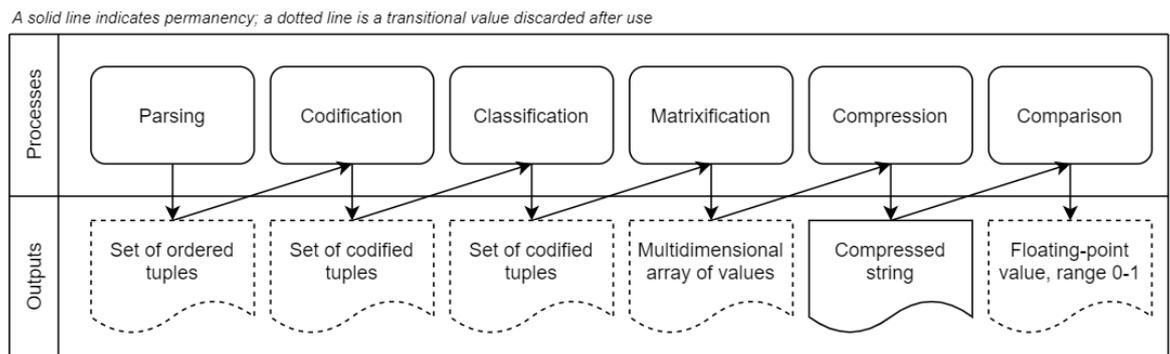


Fig. 5.3: Illustration of the alternative query representation process

Parsing: Using linear tokenisation, the query is split into distinct atomic elements. Each element is classified as either an object or an operator. Operators act upon objects and link two objects together. The produced set of object-operator-object relationships are separated into ordered pairs (tuples) such that there are a set of object-object tuples with the left-side object the object acting upon and the right-side object the object acted upon. An object may consist of other operator-object tuples; in which case the object is a unique reference to another object-object pair. Each tuple includes the operation upon the objects as a third value, so a tuple has exactly three members. Operators can include the primitives $=$, $>$, $<$, $>=$, $<=$, \neq but also include implicit operations such as ‘member of’ and complex operators such as ‘ON’ or ‘LIKE’. The type of operator is used to help in the classification process.

Formally, it is stated that this parsing process P takes as input a query Q which consists of a set of words w . A series of functions f is applied over combinations of w in Q to produce a set S of tuples t , of which each t consists of exactly three values t_1 , t_2 and t_3 – two objects, and an operator. This is shown in (1).

$$\begin{aligned} P &= \forall f(w \in Q) \rightarrow S \text{ and} \\ S &= \forall t \in S, t = (t_1, t_2, t_3) \end{aligned} \quad (1)$$

Codification: Each object and each operator are codified with a shorthand notation. All literals are then replaced with a single non-unique shorthand placeholder regardless of data type.

Formally, it is stated (2) that for all object members t (t_1 , t_2) of set S , each t_1 , t_2 is replaced with a codification of t_1 , t_2 , designated $c(t_1)$ or $c(t_2)$ (t_3 is left intact):

$$S = \forall t_1, t_2 \in S, t_1 = c(t_1), t_2 = c(t_2), t_3 = t_3 \quad (2)$$

Classification: Each object is classified as either a selection, a member, a predicate or an intersection. Each of these terms are used in their relational or set-theoretic sense; a selection is σ of values over a relation R ; a member is an element x that belongs to a set A such that $x \in A$; a predicate is a condition placed on a selection or more formally, the expression that is φ in the selection σ of values over a relation R subject to the propositional expression φ ; and an intersection is a natural join \bowtie , theta join θ , semi-join \ltimes and \rtimes , left-outer and right-outer join \ltimes and \rtimes (but not the anti-join \blacktriangleright due to the lack of a direct short analogue in SQL). The output is a temporary set that is used in the matrixification step. This set, designated K (3), consists of a distinct list of objects o and a classification c arranged as a tuple, such that:

$$\begin{aligned} K &= \{(o_1, c)...(o_n, c)\} \forall o \in S \\ &\text{where } c \in C \text{ ('selection', 'membership', 'predication', 'intersection')} \end{aligned} \quad (3)$$

Matrixification: The matrixification function f operates on set K and arranges each object on virtual X , Y and Z axes with every object appearing on both X and Y axes in every Z slice. The Z axis has a cardinality $|Z|$ of 4, consisting of a slice for each classification; selection, membership, predication and intersection. For every operator-object relationship on axes X and Y , the value at the intersections of the object and operator is marked with the value 1 only on the Z axis that corresponds to the classification of the operator on the objects within the relationship. All other intersections are marked with the value 0. The input is the codified set of tuples K that originate

from (3) above. The output is a three-dimensional matrix M which is represented as two matrices, showing axes XY and YZ (4):

$$\begin{aligned}
M &= f(K), \text{ such that:} \\
M(x) &= \text{ordered set of } \forall o \in K \text{ and} \\
M(y) &= M(x) \text{ and} \\
|M(x)| &= |K| \therefore |M(y)| = |K| \text{ and} \\
|M(z)| &= 4 \\
\text{such that the values in M consist of (XY, YZ):}
\end{aligned} \tag{4}$$

$$M = \begin{pmatrix} [0 \vee 1] & \dots & [0 \vee 1] \\ \vdots & \ddots & \vdots \\ [0 \vee 1] & \dots & [0 \vee 1] \end{pmatrix} \begin{pmatrix} [0 \vee 1] & \dots & [0 \vee 1] \\ \vdots & \ddots & \vdots \\ [0 \vee 1] & \dots & [0 \vee 1] \end{pmatrix}$$

(XY) (YZ)

Compression: The ordered matrix of objects is combined (in shorthand notation) in a string format and the resulting binary expression, read left-to-right (X), top-to-bottom (Y), front-to-back (Z) as a hexadecimal value, with Z-axis categories coded as S, M, P or I respectively. This yields a relatively short string that designated S' (5) representing the structure of the query encapsulated in M.

$$S' = \forall m \in M, \text{hex}(\text{concat}(m)) \tag{5}$$

Comparison: When query comparison is required between two queries compressed in this form, the compressed strings are used, and the Hamming distance is calculated [19] between each co-ordinate, summing these to yield a whole positive integer. Then, by inverting the resulting sum over the total population of co-ordinates, this is normalised to produce a measure of similarity in range 0-1, where 0 is completely dissimilar and 1 is identical:

In the following sections, the key components of the process are examined.

5.5 Queries as Graphs – the Query Parser and Similarity Scorer

5.5.1 Description

SQL is the de-facto language used for communicating with relational databases and is based on the well-established principles of set theory [3, 20, 21]. SQL commands SELECT and JOIN map to the relational algebra constructs. Based on these building blocks, many set-theoretic expressions can be represented as SQL queries and vice-versa. This principle underpins the matrix parser. The purpose of the matrix parser is to represent the SQL query Q as an object on which mathematical operations can be applied; given that each SQL query is essentially a narrative construction obeying syntactic rules that is later reduced to an internal representation by the query optimiser [22], some method is required to represent the query in a formal, empirically-comparable format. Text-based comparison methods do not provide sufficient support for query comparison since the key element in optimising a query is the query structure, rather than syntactic elements. Queries which are logically identical may have syntactic variations such as whitespace, alias differences and so on which introduce false negatives.

The query optimiser overcomes this issue by reducing a query to a *parse tree* [23, 24] – an internal representation of the operations and objects within a query. In PETAS, a different method is chosen for several reasons.

The first reason is that PETAS is concerned with the structure of queries rather than the binding of objects. By regarding structure over content, computationally efficient similarity comparisons can be achieved using matrix arithmetic, without the need to iterate over the process of building and comparing trees. The cost-based optimiser is adept at handling ordinary queries unaffected by the anti-patterns manifest in ORM solutions. However, using constructs like multi-layer nesting of queries (instead of JOINS) and fetching many columns increases the complexity of the query for the CBO as discovered in the literature review.

In the PETAS approach, the complexity of the query is less relevant – the cardinality of the matrices is bounded by the number of objects in the query, not solely the arrangement of these objects, which includes the relationship type. The CBO works on a query-by-query basis – if a query is not found in the cache, the query is recompiled with the attendant delay in the parse/compile/optimize process. This process becomes inefficient if numerous similar ORM-generated queries are received, so that a large majority can require compilation. PETAS looks at the query structure, notes that the query Q is (structurally) like some previous queries ($Q_1 \dots Q_k$) and infers the best schema to use without consideration of the literals in the query. Although the remapped query Q' is still passed to the optimiser, it is executing against a better-fitting schema and, where appropriate, Q' can be fitted to a previous execution plan, shortening the compilation

time and potentially undoing the anti-pattern caused by the ORM. Thus, PETAS does not seek to replace the CBO, but to conjoin the query with the best possible schema in preparation for the normal optimisation and execution process.

For these reasons, it is proposed to use an alternative method of relational query representation, based on a) identifying the relationships between elements in the query and b) describing the type of relationship, the whole to form a directed graph.

In such a representation, each object in the query (column name, or table or view name) becomes a node in the graph, and the relationship type between nodes is categorised as either:

- Membership (column name is a member of a table)
- Intersection (a relationship between two tables, typically an inner or outer JOIN)
- Predication (a condition, by way of an operator such as =, < or >, is placed on the relationship)
- Projection (the node is a subset of another node).

Although this is similar to a parse tree (an acyclic graph), it is constructed from the objects and the type of relationship they have with each other, rather than the relational operators alone, and has a completely different abstract (and internal) representation. It is also not required, unlike with a parse tree, that any binding takes place.

This directed graph can be represented in terms of the adjacency of the nodes, in a construct called an adjacency matrix, in accordance with general information theory [25, 26, 27]. More particularly it can be represented as a 3-dimensional binary adjacency matrix (termed an adjacency 'cube'), which represents the structure of the query in a 3-dimensional binary medium. Three-dimensionality is required for accuracy since it is desirable to capture the type of relationship between two objects and not simply the fact that a relationship exists. This increase in accuracy increases the utility of the similarity score and is explained further through the given examples. The i and j axes are comprised of an ordered node list, and the k axis is a type representation. NC_x is assigned to mean the node cardinality, or number of nodes, of any given cube C_x . Thus, any intersection of the three axis indicates a relationship exists and contains the value 1, and all other intersections contain 0. The result of this process is a tuple comprising of two 3-dimensional adjacency cubes, which can be represented in memory as a multidimensional array.

5.5.2 Example

Consider Fig. 5.4. This SQL query fetches a sum of sale amounts grouped by sale date and the name of the point-of-sale terminal operator. This might be a common query in a retail environment. The base tables can be modelled as an entity-relationship diagram (the logical stage of database design) using 'crow's-foot' notation in the UML style; excluding columns that are not selected for the sake of simplicity.

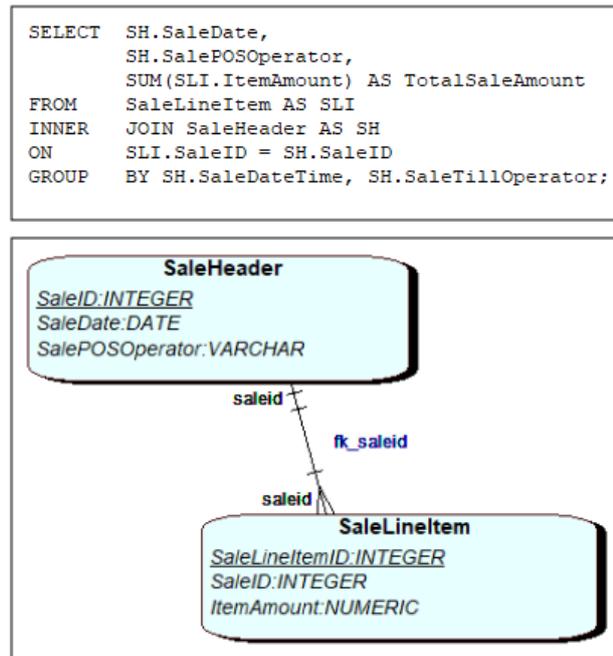


Fig. 5.4: Example SQL query with ERD diagram

Initially the same path is followed to parse this query ready for execution as existing implementations - tokenisation. Fig. 5.5 shows a list of the individual query elements from the query in Fig. 4. For this test case, a simple approach is tried; once tokenised, each token is listed along an x-axis and a y-axis, forming a square. Where one element corresponds to any other element in the SQL syntax, the value 1 is inserted at the intersection of these elements. By 'corresponds', this means 'has a relationship with'. So, for example, each column listed in the SELECT operator is linked to SELECT by virtue of being 'called' by the SELECT clause and will be marked with 1; else, marked with 0. Where there is a nested relationship i.e. SUM(SLI.ItemAmount), the nested element will only be connected to its immediate siblings on the same hierarchical level if there are any direct actions on one from the other; and to its parent, but not to its grandparent. So, SLI.ItemAmount has a relationship with SUM but not to SELECT.

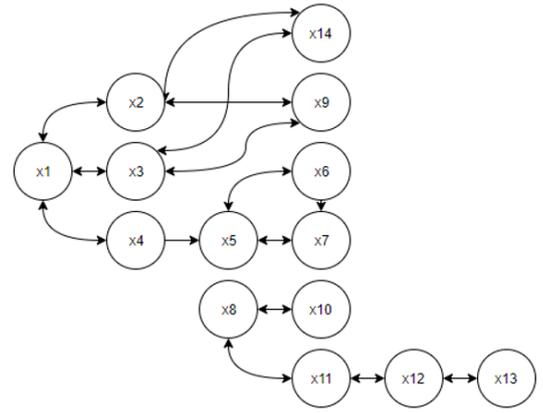
It is not important at this stage that every nuance of the query is captured; rather, that the general 'shape' of the query can be represented in some form that is suitable for computational comparison. This is a fair approach when considering that it is not possible to traverse from a bind tree to the original query in the current approach either; query translation from parsing to execution is currently a one-way operation, both in theory and in practice.

| <i>Distinct component list (excluding aliases)</i> | |
|--|----------------------------|
| Key | Value |
| x1 | SELECT |
| x2 | SaleHeader.SaleDate |
| x3 | SaleHeader.SalePOSOperator |
| x4 | SUM() |
| x5 | SaleLineItem.ItemAmount |
| x6 | FROM |
| x7 | SaleLineItem |
| x8 | INNER JOIN |
| x9 | SaleHeader |
| x10 | ON |
| x11 | SaleHeader.SaleID |
| x12 | = |
| x13 | SaleLineItem.SaleID |
| x14 | GROUP BY |

Fig. 5.5: Distinct query component list as key-value pairs

This linkage operation results in a 2-dimensional matrix with N^2 elements in a square (where N is the number of elements in the query). This is illustrated in Fig. 5.6(a). This is indistinguishable from an adjacency matrix in graph theory; an adjacency matrix lists all vertices in a graph on the x- and y- axes and indicates, through a bit field (0 or 1), whether there exists an edge between the two vertices. Thus, as an adjacency matrix has been able to be constructed, so too can a graph be constructed from the adjacency matrix which represents the query. This is illustrated in Fig. 5.6(b) for the test query. Given that the computational potential of the new model is under test, this visual graph representation only amounts to an interesting aside, but it does demonstrate how the process can neatly map from a semantic query to a mathematical, computationally-friendly construct in just a few algorithmic steps.

| | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | x10 | x11 | x12 | x13 | x14 |
|-----|----|----|----|----|----|----|----|----|----|-----|-----|-----|-----|-----|
| x1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| x2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| x3 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| x4 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| x5 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| x6 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| x7 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| x8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| x9 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| x10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| x11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| x12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| x13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| x14 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |



Figs. 5.6(a) and 5.6(b): Adjacency matrix and directed graph for the test query

Some clarification is due on mapping the query to the adjacency matrix. All relationships between elements are assumed to be undirected. Relationships exist between a column and the owning table. Relationships are pairwise, and no relationship between the main verbs (SELECT, FROM, GROUP BY) is specified using this model. Elements are associated with themselves by the property of membership (each element of a set is a member of a set containing a single element - itself, in accordance with the Zermelo-Fraenkel axiom schema of separation or e.g. by self-JOIN operations).

Fig. 5.7 shows a high-level diagram of the tokenisation algorithm.

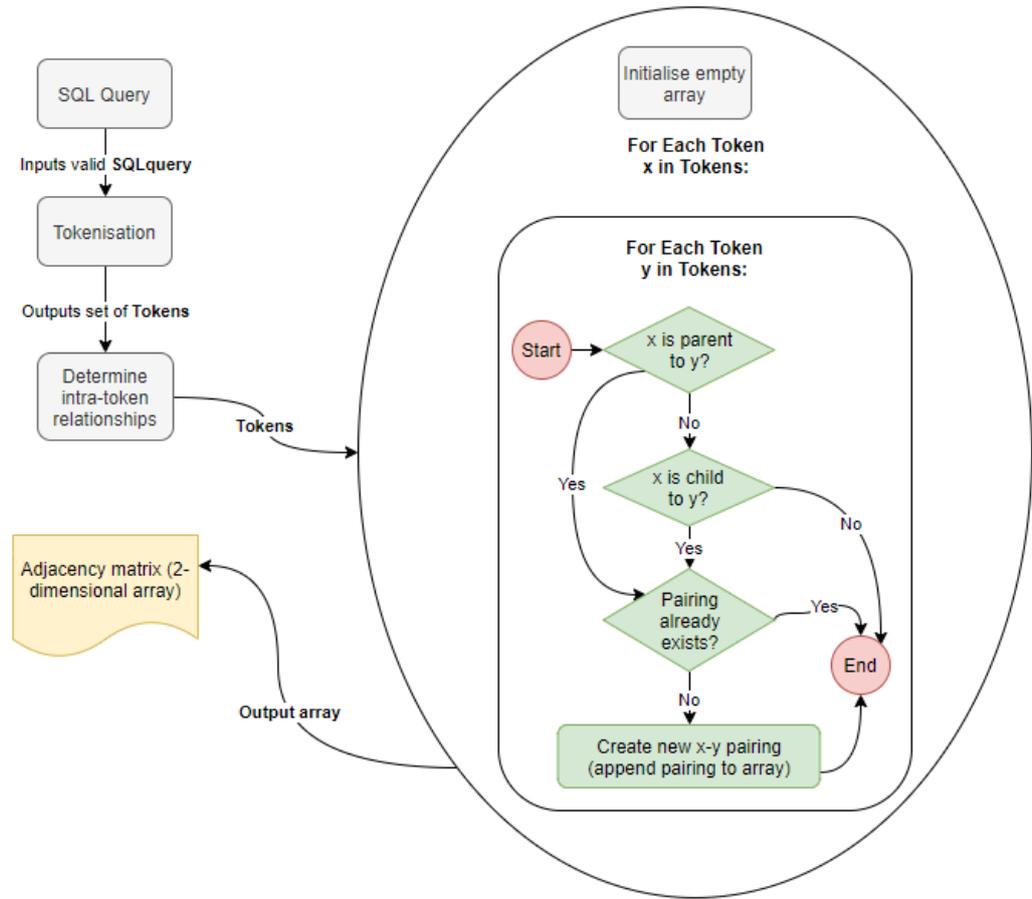


Fig. 5.7: Query tokenisation flowchart

5.5.3 Refining the algorithm

This method as described has some significant drawbacks. First, the number of vertices N of the resulting adjacency matrix will correspond to the number of tokens in the query, and so the cardinality (total number of matrix members) will always be the square of the number of vertices (N^2) and so the number of values to manage in the adjacency matrix will rise exponentially to the number of members. This is scalable for short or simple queries but this exponential increase in complexity may lead to the algorithm either breaking down as the loop-based nature of the tokenisation and relationship-inference components of the algorithm and introducing an unacceptable query parsing performance overhead during execution, in proportion to the size of the query in hand.

Next, the method of non-discrimination of the tokenisation phase leads to all tokens (elements of the query) being treated equally. This means there is no discrimination in the type of query element; tokenisation occurs and loss of detail on whether an element is an actor (e.g. a keyword

like SELECT) or an object being acted upon (e.g. a column/attribute). Considering that the object goal is to be able to represent the query as a 'shape', represented by a graph, for the express purpose of being able to compare some query 'shape' to some other query 'shapes' to determine a) the likelihood or feasibility of execution plan re-use and b) for the benefit of decision-making when selecting a schema, then the ability to differentiate between at least actors and objects would re-introduce a major degree of detail lost during the initial attempt at query modelling.

Finally, the practice of non-discrimination between token types means that the corresponding loss of detail necessitates the provision of an accompanying legend or map, mapping each token to its columnar position or label. In the relational model, such maps are known as attribute headings and carry domain information. Not to do this would introduce a higher probability that two queries are compared and incorrectly categorised as similar, despite the token mappings being very different. Conversely, minor differences between two otherwise-similar queries such as the rearrangement of a JOIN or the use of a CTE (common-table expression) instead of a subquery, may lead to the incorrect categorisation of two queries as being largely dissimilar despite their structural similarity.

These flaws can be addressed by introducing a third dimension to the adjacency matrix. At present, the mechanism is to determine whether two tokens have a relationship – the type of relationship is not being determined. If relationship type was being mapped along a third dimension (z-axis) the adjacency matrix is transformed into an adjacency cube. But how to distinguish type? In the two-dimensional model, a relationship is specified as a 'has-a' or 'is-a' or 'a dependency exists upon' - the three terms amounting to the same definition, that is a binary choice between whether a relationship exists between two elements. In the three-dimensional model, the type of relationship is also under consideration.

To determine the allowable domain of types, the SQL syntax and some of the underlying relational theory can provide a solution. Consider that the solution now differentiates between the thing being acted upon (the object) and the thing doing the acting (the actor), then an intuitive modelling method might be to list all the objects on the x- and y- axes, and classify the actors into a range of types along the z-axis. What relational operations are available to help provide some taxonomy for classification? The relational algebra provides several clues. Projection, which maps broadly to the SQL equivalent of SELECT (it is not the same as relational selection); union; intersection (which maps to several types of JOIN, both relationally and in SQL); rename, which can be actualised in SQL using aliasing; filtering (in SQL, rendered as JOIN clause predicates and WHERE clauses and relationally, a selection); amongst others. Interestingly, there is no specific relational form for aggregation, but this facility exists in SQL. Aggregation can be rendered in the relational algebra, albeit awkwardly.

Given the basic relational operators, these can be roughly classified into five distinct categorisations. Note that this still does not render a complete representation of the query in an

adjacency cube format; the loss, notably, includes relationships between elements and groups of elements; and the particulars of primitive operators; but the ability is gained to capture additional information about the query through the use of the z-axis. Consequently, the four chosen categories are projection, intersection, membership and predication which together cover a large proportion of the legal operations in relational algebra and SQL syntax. Each category occupies a row on the Z-axis. These terms are defined below.

Projection is the relational-algebraic term for the exposure of some relation R (or relational operation) on one or more other relations, limiting the attributes exposed to some subset of R such that the attribute values are limited. Thus, any element can be identified that is SELECTed within the SQL query FROM some R being projected, and at the intersection between the attribute on the x- and y- axes in the adjacency cube and the projection layer of the z-axis, the value 1 may be inserted. An intersection of R.a1 and a1 on the projection layer of the z-axis ($z = 0$) indicates R.a1 is projected (SELECTed) from R and so there is no need to include the actors on the x- or y- axes. This has the benefit of reducing the number of components on the x- and y-axes and improving scalability of the model.

Intersection is next defined as the case where R is intersected with some S (both R and S being relations/relational variables); such that the intersection (term not used solely in the strict relational sense) causes some form of a SQL JOIN. This JOIN will either have an additive, subtractive or null effect on the attributes being returned (and on the range of data returned), depending on what attributes are projected. So, for example, the relational semijoin [left-bowtie] or [right-bowtie] corresponds somewhat, but not entirely to, the INNER JOIN with predicates in SQL. Therefore, in the adjacency cube any intersection of an element where there exists a JOIN directly upon another element is marked with 1. As JOINS in SQL are actioned between tables (relations/relational variables) and not columns (attributes), this, by definition, means intersections in the adjacency cube may only exist between relations and not attributes. Note that the JOIN predicates are not lost but are captured in the predication layer and not the intersection layer. Also note that should a true relational intersection occur - (a natural JOIN, and represented in SQL by the INTERSECT keyword) then this can also be represented in this layer.

Membership is defined as an attribute such that an attribute a is a member of a relation R if the attribute is present as an instance of a domain in all the tuples in R. In SQL parlance, it is enough to say a column is present in some table (or some expression of a relation such as the join of two tables). This layer differs from the projection layer as all columns in the query, regardless of whether they are SELECTed, are mapped to their relation in this layer. This is particularly important in WHERE clauses and JOIN predicates, for example, which may impose restrictions on the data being returned without necessarily returning that data in the result set. Membership is established between an attribute and a relation (but R is always a member of R, if $R = R$), and

attributes are members of themselves (an is a member of an if $an = an$) - more formally, relations are improper subsets of themselves, as are relations, in the adjacency cube.

Finally, predication is the layer that deals with the SQL clauses which compare one element to another. Predicates are found in the WHERE clause, in JOIN predicates and in more complex constructs such as LIKE, IN and CASE statements. In their basic form these are two elements separated by a primitive operator (such as $=$, $>$, $<$). Relationally, these are filters and projections that are limited by filters are selections.

In the adjacency cube, all predicates are taken as pairs of values under comparison and the intermediate operators are discarded. If any constants are involved, these are mapped to placeholder values and included in the dimensions of the cube. Each placeholder value is notated as $p1 .. pn$. For the example ' $...WHERE R.a1 = 7 AND R.a2 < 5$ ', $(R.a1 = 7)$ becomes $p1$ and $(R.a2 < 5)$ becomes $p2$. Then any relationship between the predication and another element is expressed using a 1 at the intersection in the normal manner.

In Chapter 6, this query representation idea is further described through example. Chapter 6 also presents algorithms for implementation, describes the implementation with code examples, describes the testing process and presents the results, implemented in the RDBMS PostgreSQL.

5.5.3 Calculating the similarity between three-dimensional adjacency cubes

Given an adjacency cube as an output from the query parser process, the cube is compared to previous adjacency cubes stored in order to establish, from the performance history, the best schema allocation for the cube based on the allocations of similar cubes. Given cube C_1 , a second cube C_2 is fetched (being a cube from memory, or a cache) for comparison. This comparison may be repeated many times.

The similarity scoring process takes two cubes as inputs and constructs a third cube C_3 based on the respective Hamming distances [19] (defined as the number of transitions required to get from state A to state B in a number system, in this case base 2 integers) between each corresponding intersection - that is, each intersection of nodes and type. Each corresponding intersection ($[i, j, k]$) is compared between cubes, the Hamming distance forming the third cube. When $NC_1 \neq NC_2$, the cube with the lowest cardinality, $\min(NC_1, NC_2)$ is aligned to any corner of the larger cube then padded with 0s. The reason for the padding is to ensure the cubes are of the same size; formally, to ensure $NC_1 = NC_2 = NC_3$. This is important since otherwise an error is introduced to the outcome of the Hamming distance, directly proportionate to the disparity in cube size. Any member of the population of C_3 can then be calculated using (6):

$$C_{3i,j,k} = (C_{2i,j,k} - C_{1i,j,k})^2, \forall ([i, j, k]), \quad (6)$$

where integers i and j , representing the nodes, are bounded by the conditions $i = j, j > 0, j \leq NC_1$, and $j \leq NC_2$. Integer k , representing relationship type, is bounded by $0 < k \leq 4$. Using (1) for all distinct $([i, j, k])$ co-ordinate triples in C_1 (overlaid on C_2), one may then calculate the similarity score by summing the Hamming distances at each resulting $([C_3^i, j, k])$ intersection in the third cube to calculate an integer S , defined as (7):

$$S = 1 - \left[\frac{(\sum C_{3(i,j,k)}) / 2}{|C_3|} \right], \quad (7)$$

which normalises S so that S is bounded by $0 \leq S \leq 1$.

This process is illustrated with the following example. Consider Fig. 5.8, which describes query Q , another query M , and how the structure of these queries can be represented using directed graphs:

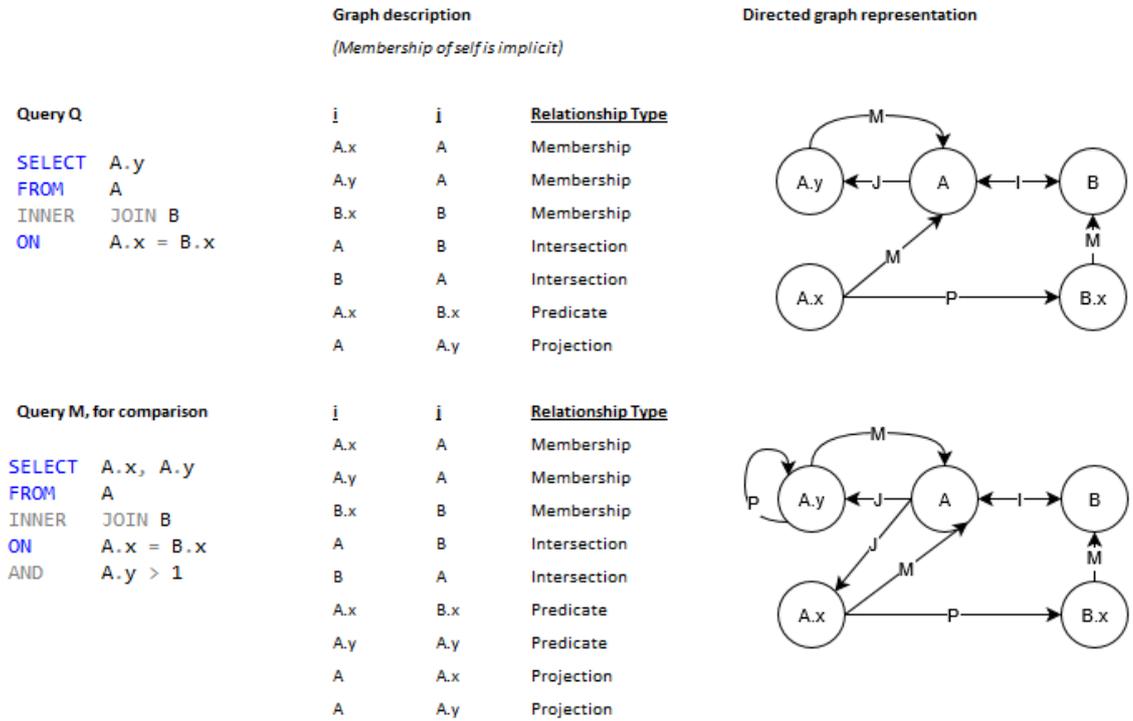


Fig. 5.8: Directed graph representations of SQL queries

These graphs look similar but are not identical. Q has an extra projection (in the SELECT, A.x) and an extra predicate (in the WHERE, A.y > 1). The conclusion from the examination of both queries, subjectively, is that the queries are 80% similar, therefore assign $S = 0.8$.

This can then be checked by calculation using the matrix parsing method. Let the directed graphs first be represented using 2-dimensional adjacency matrices. The application of (6) to the adjacency matrices for Q and M obtains the matrix of the Hamming distances, C_3 as shown in Fig. 5.9:

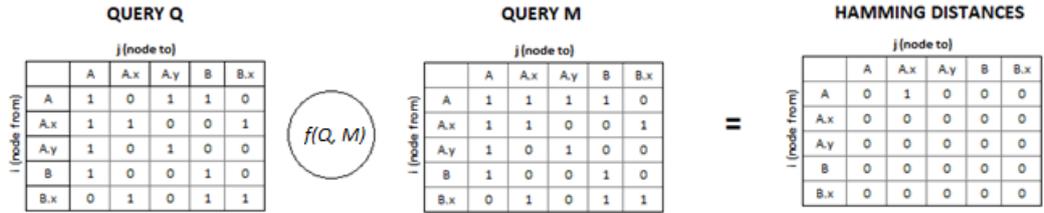


Fig. 5.9: Calculating Hamming distances from adjacency matrices

It is noted that $H = 1$ (there is only 1 non-zero value in the resulting C_3 matrix) and $NC_3 = 5$. Equation (7) is used to normalise this, and the result is $S = 0.96$. This is a significant deviation from the initial subjective estimate of 0.8 and implies there is only a 4% difference between the queries. This is counter-intuitive since there are 2 significant differences in the query structure from a total of just 5 objects. Note that A.y features as the object of both differences, but C_3 shows only one deviation. The information about the second deviation is lost when using 2D representation, since the information about the type of difference is not taken into consideration.

The matrices are now recalculated, but in three dimensions – that is to say, to recalculate the adjacency cubes, to preserve this type information on the Z axis. Fig. 5.10 illustrates the exploded cubes and the subsequent cube C_3 that results from the application of (6):

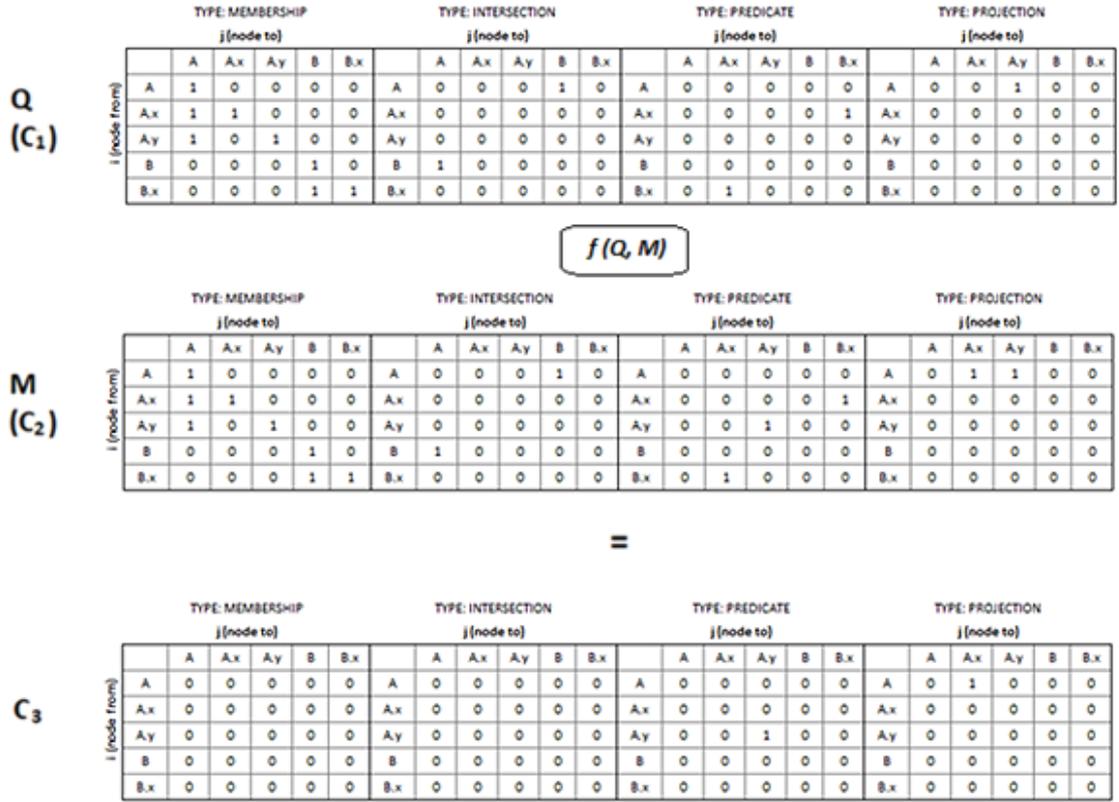


Fig. 5.10: Calculating the adjacency cubes and Hamming distances for Q , M

Recalculating (7) on C_3 , NC_3 remains at 5, but H is now the sum of non-zeros in C_3 , so $H = 2$. This yields $S = 0.84$. The type information has not been lost, and the similarity score calculated is now much closer to the original subjective estimate of 0.8.

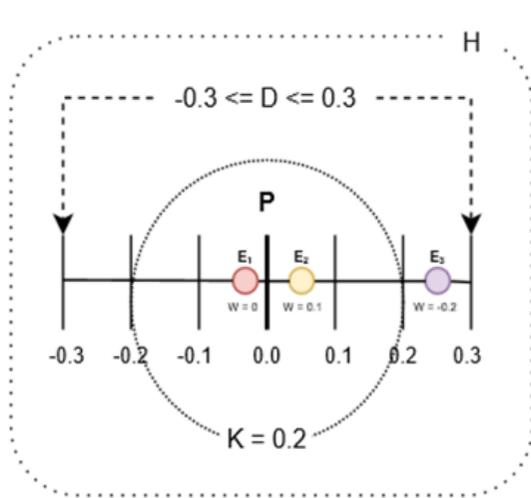
It is acknowledged that, using this method, the node identifiers are discarded, and that such similarity comparisons may therefore detract from accuracy due to factors like table population (cardinality). However, this reflects the approach used in execution plans, which are constructed indirectly from the parse tree structure of the query [22, 28] therefore, structure has been shown to be significant in query tuning and considerations like database statistics are regarded as secondary in this context. It is also acknowledged that, as described, the process will not support certain set operations such as UNION; the example presented here represents the initial implement of PETAS and future versions will deal with constructs such as aggregates. There is also the scope to use a 4th, 5th or n th dimension to further describe the query, however this will come at the cost of exponential computational effort since the number of calculations scale exponentially.

5.5.2 KNN selection – query ranking using K-nearest-neighbour

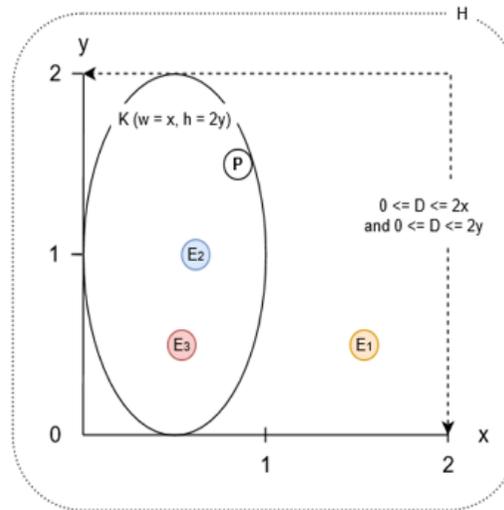
K-nearest neighbour (KNN) is a machine learning classification method that can classify a point P in relation to entities $(E_1 \dots E_n)$ in a domain D of multi-dimensional Hilbert space H (a plane occupying N dimensions) based upon the proximity of P to a set of K entities $(E_1 \dots E_n)$ as modified by a weight W assigned to each $(E_1 \dots E_n)$ (modification can be additive or multiplicative). The boundary, K , can be defined in two ways – either whether K enables a binary classification according to if any given E_i is a neighbour of P with the outcome 0 or 1 depending on whether E_i falls within the boundary K , or by simply including the nearest neighbours of P by some distance measure (Euclidean, Manhattan etc).

The concept of using a boundary function for K is illustrated in the figures below for $N = 1$, $N = 2$ and $N = 3$. It is notable that K need not be linear or a constant, but is bounded only by the inclusion of P and some upper limit – Fig. 5.11(a) shows K as a circle on the number line with radius k ; Fig. 5.11(b) shows K as an ovoid with height $1.8y$ and width x (foci at $[0.27y, 0.5x]$ and $[1.73y, 0.5x]$); Fig. 5.11(c) shows K as a cuboid occupying space $x, 1.5y$ and $2z$. All W_x are additive in these examples. K can be arbitrary and variable.

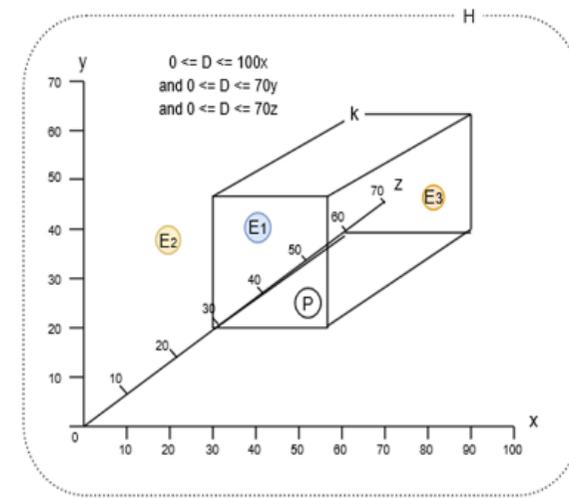
Fig. 5.11(a) shows how entities E_1 and E_2 fall within K , but E_3 does not. Regardless of dimensionality, each E_x is affected by some weight W_x . Adjustment of each W_x can therefore affect the position of E_x and membership of K . Thus, the population of E_x points inside K varies over successive feedback cycles through the ‘movement’ of $(E_1 \dots E_n)$ within D . This trait can be used to affect the outcome of classifications and subsequent schema assignments.



| Entity | Value | Weight | Position | Within K? |
|--------|-------|--------|----------|-----------|
| P | 0 | 0 | 0 | Y |
| E1 | -0.03 | 0 | -0.03 | Y |
| E2 | 0.04 | 0.01 | 0.05 | Y |
| E3 | 0.45 | -0.2 | 0.25 | N |



| Entity | Value | Weight | Position (x, y) | Within K? |
|--------|----------|--------|-----------------|-----------|
| P | 0.8, 0.5 | 0 | 0.8, 0.5 | Y |
| E1 | 1.2, 0.2 | 0.3 | 1.5, 0.5 | N |
| E2 | 1.6, 1.9 | -0.9 | 0.7, 1.0 | Y |
| E3 | 0.5, 0.4 | 0.1 | 0.6, 0.5 | Y |



| Entity | Value | Weight | Position | Within K? |
|--------|------------|--------|------------|-----------|
| P | 47, 20, 30 | 5 | 52, 25, 35 | Y |
| E1 | 39, 39, 39 | 1 | 40, 40, 40 | Y |
| E2 | 20, 38, 20 | 0 | 20, 38, 20 | N |
| E3 | 90, 55, 80 | -10 | 80, 45, 70 | Y |

Figs. 5.11(a), 5.11(b) and 5.11(c): K in one, two and three dimensions

For PETAS, an alternative approach is used, nearest- K number of E_x rather than all E_x within a K boundary. In PETAS, this is a one-dimensional KNN representation. However, instead of a boundary condition defining K , the K -nearest to point P is selected. For example, $K = 3$ means to include the 3 nearest neighbours to point P . This still enables the movement of each E_x within the domain but provides the advantage of being able to sort all E_x in an ordered list and simply select the top K from the list. It also avoids having to set a fixed boundary condition for K which may be suboptimal. Note that the term P is replaced with Q , since Q is the point of origin for the KNN with a value of 1, and E_x is replaced with Q_x , since the entities in the generalisation are queries in PETAS.

To proceed, some query Q is taken and the adjacency cube is calculated. Then, for some predefined number of queries ($Q_1 \dots Q_x$) together with their weights ($W_1 \dots W_x$) drawn from a ‘metadata cache’ of training data, Equation (1) is used to produce C_β for each (Q, Q_x), and Equation (2) is used to produce S'_x , the similarity score, from C_β . The term S'_x is multiplied by the weight W_x resulting in S_x and this value is stored as an addition to the tuple (Q, Q_x, C_x, W_x). By this repetition over x members of the metadata cache, x tuples of (Q, Q_x, C_x, W_x, S_x) are obtained. The S_x values are plotted along the number line and so the outcome is a list of x values of S_x , each S_x associated with an Q_x , and falling in the domain $0 \leq D \leq 1$. Weights are updated after the schema selection process.

The output of this process is the top K queries and their associated schema assignments on this number line closest to Q , which are used when moving on to the schema classifier.

5.6 The Schema Classifier and Query Mapper

5.6.1 Description

The outcome of the KNN process leads to the identification of K tuples from the training data in the metadata cache. Each tuple consists of the unique identifier of the query in hand, Q and the unique identifier of the identified neighbouring query, Q_x identified by the KNN selector. Each tuple also has as a prior schema classification C_x associated with it - this is the identifier of the schema on which the Q_x query last ran, and a weight W_x . Thus, each tuple has the construction (Q, Q_x, C_x, W_x). From these identifiers, a majority verdict for schema choice can be attained by examining all the C_x values associated with the Q_x values in the tuples and applied to the query Q in hand. In this way, Q is assigned the most ‘popular’ schema choice of all its nearest-neighbouring queries (queries with greatest structural similarity). Note that the conditions ($k \bmod 2 \neq 0$) and ($k > 1$) should be true to ensure a majority. Next, Q is mapped syntactically from the base schema

for which it is written to the majority verdict schema and passed to the RDBMS query processor to execute and return the result set. In the proof-of-concept implementation, a stored procedure was written to execute this but in a full implementation, a lower-level structure for re-mapping, such as the parse tree, would be used.

After execution, the execution duration for the query Q , termed d , is returned; this information is used to compare against the previously-recorded d of each of the neighbouring queries ($Q_1 \dots Q_k$), designated $Q_x d$. For each Q_x , if $Q_x d > d$ then the schema choice C_x for the query Q_x was deemed ‘useful’ – this means query Q executed quicker than Q_x , and the corresponding weight W_x is increased (one could equally measure CPU cost, or I/O consumption instead of query execution time). Conversely, if $Q_x d < d$ then C_x for the query was not useful – this means query Q executed slower than Q_x , and the weight W_x is decreased. If $Q_x d = d$ then W_x is unchanged. In this way, the process rewards each Q_x query in the metadata cache with an increased weight W_x according to whether the schema choice of the query was a good choice, in the sense that Q executed faster than Q_x , for the query Q . Successive iterations mean that the Q_x queries whose schema choices are most applicable to the recent inbound flow of Q queries are probabilistically more likely to be selected than those Q_x queries which have schema choices leading to slower executions of Q than Q_x . By ‘probabilistically more likely’, it is meant that due to the application of higher weights W_x to the most-used queries in the metadata cache, these increased weights for each ($Q_1 \dots Q_k$) in the KNN calculation make these queries more likely to be selected as neighbours to Q than any other query in the cache. With many iterations (hundreds, thousands or tens of thousands) the queries in the cache that are most useful gain the largest weights, and the least useful are rarely if ever selected – these are pruned periodically by the asynchronous cache management process. Thus, the metadata cache of previously-run queries continually adjusts itself to the inbound flow of queries Q , and any change in the general structure of the query flow is soon reflected in the classifier.

5.6.2 Feedback mechanisms

Database queries are highly variable. This variability can range from intra-query, where one query is unlike the next, to variation caused by different query patterns being generated by different applications (e.g., ORM vs. stored procedure calls), to long-term query pattern changes over time as ORM solutions are upgraded. Creating a classifier based on some N number of different query types is undesirable in these circumstances, firstly because creating such a system is resource-intensive and must act as a catch-all for unknown query flow. These limitations are experienced in expert systems which are more rigid than systems that can respond to external stimuli.

Consequently, PETAS uses a machine-learning approach, classifying queries but having in place

processes for improving the accuracy of the classifier based upon the success of its own previous classifications. These are feedback mechanisms.

Three feedback mechanisms are proposed in PETAS. The first, previously described, updates the weights in the metadata cache as queries are executed. The second is the feedback mechanism for the KNN process. This is run asynchronously, i.e., not within the execution timeframe of the query, and is responsible for a) inserting the most recently run query Q into the metadata cache (with a weight = 1) and b) pruning the metadata cache of entries based on the lowest weights and the most aged entries. In this way, the metadata cache population is managed. It is proposed that a future iteration of PETAS will include a third feedback mechanism to reduce the need for multiple instances of schemas. In the current proof-of-concept implementation, there is a requirement to maintain multiple instantiations of schemas to enable schema choices.

This has the effect that data is duplicated and requires, at minimum, a doubling of storage space for two schemas. This is currently necessary since existing RDBMS platforms do not support the creation of logical-only schemas at the whole-database level. It is envisioned that PETAS could be used in a multi-schema environment, where there exists a 'base' schema of the physical data and multiple alternative schemas which contain arrays of logical pointers to the data in the base schema. These pointers will also consume space but could be designed in such a way as to occupy less space than the physical data. These alternative schemas can then be created and destroyed by an asynchronous process which is responsible for a) designing and implementing schemas based on the flow of inbound queries, b) assessing existing schemas against how often they are executed against and c) destroying under-utilised schemas.

The similarity scoring mechanism, including details of the KNN mapping process and the identification of a suitable schema derivation for use by a given query, is described more fully in Chapter 7, which also provides algorithmic implementations; code details from the practical implementations using Python and PostgreSQL; the experimental design and results from testing, before discussing benefits, drawbacks and overall success of this approach in a life-like environment.

5.7 Dynamic Schema Redefinition

5.7.1 A new definition of query efficiency

Query performance can be measured in many ways, often dependent upon the platforms themselves. For example, cloud platforms such as Microsoft Azure use Database Transaction Units (DTUs), a blended measure of CPU and I/O use, to measure query throughput [29]; natively, cost can be measured per-query as a ‘query cost’, a purely relative measure that has roots in CPU use and I/O load [30]; others may prefer to define query performance, and therefore RDBMS efficiency, as a combination of more traditional system administrative measures such as disk reads, disk faults, CPU use and memory used.

However, there is no ubiquitous definition of query efficiency, and so it is necessary to invent one for the purposes of comparative analysis before proceeding. In this section, a new definition is proposed, and it is shown through example how the reduction of the number of rows of data that the query execution engine must traverse for a given query is correlated to the cardinality (number of rows) of the data set in question, thus demonstrating how query efficiency can be defined as a ratio of rows required by the query versus row cardinality of the relation being queried. The relational algebra is used. This efficiency metric is not central to the novel contribution to knowledge, being a simple definition of a measure to enable further testing on dynamic schema definition, and so is not discussed elsewhere.

When defining the efficiency E of a query Q , it is first stipulated that this query must be a selection, since the efficiency relates to the rows selected versus the rows available (8). The term E is then defined as the ratio of data values returned from the relation R by the selection σ , as modified by some predicate φ , divided by the number of data values read by the query (using the cardinality notation $| \cdot |$) to return the query result (9):

$$Q = \sigma\varphi(R) \quad (8)$$

$$E(Q) = |Q| / |\sigma\varphi(R)| \quad (9)$$

The number of data values read is chosen as a measure since the purpose of this approach is to limit the number of data accesses required to service a query towards an efficiency ratio of 1; to this end, this measure is not concerned with CPU thread efficiency, memory use, network use or other measures individually. For simplicity, a read is defined as a collection from storage of a

single row/column intersection value by the RDBMS rather than as an operation to collect a specific number of bytes, since the number of bytes returned by a read can be variable, but if this figure is known, then the subsequent calculation is straightforward.

The number of reads required to service a query are determined by several factors; the type of components in the query execution plan, the cardinality of the tables (relations) involved as query sources, and the availability of suitable views and indexes on the base schemas. In other words, the efficiency is hereby defined by the amount of *necessary* data reads required to return the result versus the amount of *unnecessary* data reads that took, or would take, place. The definition of efficiency measure E can therefore be precisely defined, differentiated by the four common types of query plan component that read data [31, 32, 33, 34], as follows. As in (8) and (9), cardinality (number of rows) of a relation is notated using the standard notation $|R|$ and the notation described in Codd's relational algebra [20] is employed.

For *table/index scans*, which involve reading the whole base relation or clustered index R and extracting results based on predicate φ , it is assumed that the yield of the query can be retrieved entirely from the scan; else, the efficiency E must be distributed according to the cost of the individual components in the query plan as appropriate. The attributes of a relation are denoted as $R(a_1 \dots a_n)$.

The efficiency of a table or index scan against R can therefore be defined as shown in (10):

$$E(\sigma\varphi(R)) = (|\sigma\varphi(R)| \cdot |\sigma\varphi(R(a_1 \dots a_n))|) / (|R| \cdot |R(a_1 \dots a_n)|) \quad (10)$$

Equation (10) can be illustrated with a worked example.

If query Q yields 50 rows of 9 columns from the RDBMS, then:

$$|\sigma\varphi(R)| = 50 \text{ and } |\sigma\varphi(R(a_1 \dots a_n))| = 9.$$

If $|R|$ could yield 10,000 possible rows of data, with $|R(a_1 \dots a_n)| = 11$ possible columns, then these values are substituted as:

$$(50 \cdot 9) / (10000 \cdot 11) = 450 / 110000 = 0.004.$$

This means that, implemented using a single table scan, the efficiency $E(\sigma\varphi(R))$ of query Q has an efficiency ratio of 9/2200, or 0.4%; the scan operation read and discarded 99.6% of all data in R to

service the query $Q = \sigma \varphi(R)$, assuming that a read is defined as an I/O operation on a single row/column intersection. This idea scales linearly to the definition of a read as a row read, since each value in a row of data would need to be read to include or exclude it from the query result, given no key is used in table or index scans.

Comparing this against *B+-tree index seeks*, which involve reading a defined ordering of the base relation R structured as a B+- tree and extracting results based on some desired predicate key K, the efficiency correlation is calculated based on the number of tree traversal operations required and the reads required to find the appropriate data in the leaf level of the tree. As with the table scan, it is assumed that the yield of the whole query $\sigma K(R)$ can be retrieved from using the predicate key and from within the index; where this is not the case, the efficiencies will need to be distributed across the relative components in the query execution plan according to component cost. The number of traversal operations (T) depends on several factors: the average row length (L), the number of rows per leaf page (RP), the standard page size (S) in the RDBMS, the number of rows in the relation R (denoted |R|) and consequently the number of leaf-level pages required (|P₀|) in the index.

Using the methodology for calculating read operations on a B+-tree described by Delaney [8], the average rows per page and the number of leaf pages in a B+- tree can be derived using the following equations, expressed as (11) and (12):

$$\begin{aligned} \text{Average Rows per Page (RP)} = \\ \text{Standard Page Size (S)/Average Row Length (L)} \end{aligned} \tag{11}$$

$$\begin{aligned} \text{Number of leaf pages (|P}_0\text{|)} = \\ \text{Number of Available Rows (|R|)/Average Rows per Page (RP)} \end{aligned} \tag{12}$$

By calculating |P₀|, it is now known how many leaf pages are required, and consequently the number of intermediate level pages required from |P₀| can be inferred by looking at the bytes required (B) by the datatype of the key for a single value; for a single, non-composite integer column this is normally 4 bytes, for example. The structure of these pages varies between RDBMS implementations, but in Microsoft SQL Server, to illustrate, the page pointer length (PP) is 6 bytes and row overhead (RO) is 1 byte, yielding 11 bytes for a single intermediate-level row in the B+- tree ([8], pp. 322) with a single integer key, although this may vary between RDBMSs and software versions. The number of intermediate-level pages (|P₁|) required can be calculated as per (13):

$$|P_1| = (|R| \cdot (B + PP + RO)) / S \tag{13}$$

The calculation now progresses to the next level of the index (which can be either another intermediate level or the root level). This level contains page(s) with pointers to the previous intermediate level of pages. Therefore, only enough pages in this level are needed to contain the pointers to all pages in the previous level, as illustrated in (14):

$$|P_2| = (|P_1| \cdot (B + PP + RO)) / S \quad (14)$$

This calculation holds true for all intermediate and root levels, so the total number of pages required can be calculated as the sum of all pages across all levels, and consequently the total storage required (TS) in bytes for the index as this figure multiplied by the standard page size for the RDBMS, as per (15):

$$TS = S \cdot \Sigma(|P_0|, |P_1| \dots |P_x|) \quad (15)$$

Having calculated the various values of P_n , the next calculation is the average number of traversal operations T as shown, starting from the root level (denoted P_x) through to the leaf level (denoted P_0), subtracting iteratively from x until 0 is reached, culminating in (16):

$$T = \{ \forall x > 0, \lceil (|P_{x-1}| / 2) \rceil + \lceil (|P_{x-2}| / 2) \rceil + \dots \lceil (|P_{x-n}| / 2) \rceil \} + \dots \lceil (|R| / |P_0| / 2) \rceil \quad (16)$$

Table 5.12 illustrates an example using these formulae, which shows 28 reads are required against a B+-tree index containing 10,000 rows ($|R|$) where each row is on average 200 bytes (L), given some standard RDBMS parameters (S , B , PP and RO), deriving the remaining variables from these parameters (RP , $|P_0|$, $|P_1|$, TS and finally T):

Table 5.12: Worked example for calculating traversal cost across a B+-tree index of 10,000 rows, consisting of a 200-byte average length per row.

| Description | Page size (bytes) | Avg. row length (bytes) | Avg. rows per page | Num. of rows in relation R | Num. of leaf pages in index | Bytes required per index key |
|-----------------|-----------------------------|-------------------------|------------------------------|-----------------------------|-----------------------------|---------------------------------|
| <i>Notation</i> | S | L | RP | R | P ₀ | B |
| <i>Value</i> | 8096 | 200 | 40.48 | 10000 | 248 | 4 |
| Description | Page pointer length (bytes) | Row overhead (bytes) | Num. of pages in first level | Num. of pages in root level | Storage required (bytes) | Traversal cost (reads required) |
| <i>Notation</i> | PP | RO | P ₁ | P ₂ | TS | T |
| <i>Value</i> | 6 | 1 | 14 | 1 | 2,096,864 | 28 |

Finally, the efficiency E of searching any B+-tree index can be defined as the ratio of the number of reads required across some index R to read one row of data.

$\sigma K(R)$ can then be defined as the inverse of T for one row of data, as per (17):

$$E \sigma K(R) = \frac{1}{\{\forall x > 0\}, \lceil (|P_{x-1}|/2) \rceil + \lceil (|P_{x-2}|/2) \rceil + \dots + \lceil (|P_{x-n}|/2) \rceil + \dots + \lceil (|R|/|P_0|/2) \rceil}$$

(17)

The index in the worked example shows T = 28, so 28 row reads are required, on average, to find one row identified with a key using the example variables (200 bytes/row, 10,000 rows).

Using this example,

$$E \sigma K(R) = 1/28 = 0.036 = 3.6\% \text{ efficiency (for this case).}$$

This can be modified for multiple rows N by changing the numerator to N accordingly.

Compare this against the values for a simple table scan produced by (17), substituting MAX(a_n) = 3 as a reasonable assumption of column count for both the query and the available columns (any integer substitution of MAX(a_n) will do, if $\sigma \varphi(R(a_1 \dots a_n)) = |R(a_1 \dots a_n)|$, as they cancel), as shown in (18):

$$E(\sigma\varphi(R)) = (|\sigma\varphi(R)| \cdot |\sigma\varphi(R(a_1 \dots a_n))|) / (|R| \cdot |R(a_1 \dots a_n)|) =$$

$$E(\sigma\varphi(R)) = (1 \cdot 3) / (10000 \cdot 3), \text{ therefore}$$

$$E(\sigma\varphi(R)) = 3 / 30000 = 1 / 10000 = 0.01\% \quad (18)$$

With a table scan producing an efficiency ratio of 1/10000 (0.01%) and a B+-tree index scan producing an efficiency ratio of 1/28 (3.57%), it is clear that a) the index seek is more efficient for this example and b) that both methods fall short of the goal of full query efficiency with a ratio of 1/1.

It is clear that indexes, although useful in reducing the search space, still require unnecessary traversal through data unrelated to the query and as such, a smaller search space and consequently better query efficiency would be beneficial in reducing the number of required reads regardless of whether indexes are used. It is acknowledged that this example has been simplified to read key values rather than whole rows, but the number of traversals remains the same in either case.

In Table 5.13, the importance of reducing |R| is illustrated by modelling the relative efficiencies using this method for a range of queries using a single index seek, varying the average row length RL at various intervals between 10 and 3,200 bytes and the number of rows in the relation |R| between 100 and 1,000,000 at logarithmic intervals, using a 3-level index, calculating the efficiencies using Equation (2):

Table 5.13: Relative index seek efficiency for varying conditions using a simple efficiency measure

| Index seek efficiency | | Available rows (R) | | | | | |
|-----------------------|------|----------------------|----------|----------|----------|---------|---------|
| | | 100 | 1000 | 10000 | 100000 | 1000000 | |
| Bytes per row (L) | 10 | 0.2470% | 0.2466% | 0.2466% | 0.2430% | 0.2115% | 0.0922% |
| | 100 | 2.4662% | 2.4296% | 2.4296% | 2.1154% | 0.9224% | 0.1389% |
| | 200 | 4.9242% | 4.7803% | 4.7803% | 3.6991% | 1.1341% | 0.1429% |
| | 400 | 9.8155% | 9.2598% | 9.2598% | 5.9124% | 1.2812% | 0.1450% |
| | 800 | 19.5010% | 17.4236% | 17.4236% | 8.4363% | 1.3700% | 0.1461% |
| | 1600 | 38.4921% | 31.1590% | 31.1590% | 10.7256% | 1.4191% | 0.1467% |
| | 3200 | 75.0224% | 51.4311% | 51.4311% | 12.4093% | 1.4451% | 0.1469% |

It is evident from the data that although index seeks are efficient when |R| is relatively low and L is relatively high, this efficiency quickly tends to LIM \rightarrow 0 as the number of rows in the index grows, and generally improves as a function of the number of bytes per row. This indicates that the primary driver of efficiency, as defined, is the number of available rows in |R| and strengthens

the case for data structures which are tailored to the query, reducing $|R|$, in order to reduce the amount of data required to be traversed to yield a query result, thereby maximising efficiency.

For *bookmark/row lookups* (against either indexes or heaps, given a page number and row offset), this is defined as a single read of a row given the precise physical location is known and therefore, under the simplified definition of a read, is 100% efficient.

The findings in this area strengthens the case for the reduction of row data for query traversal, lending credence to the idea of subset schemas as alternative query sources.

This idea, forming the central paradigm in the proposed solution, is explored further in the next section.

5.7.2 *Dynamic schema redefinition process design*

As a result of the work described in Section 3, the denominators in common across all types of read operation are clearer. These are the total available values in R ($|R|$) that must be addressed to produce the result set of $\sigma\varphi(R)$ or $\sigma K(R)$ (given there is no purpose in differentiating between table/index scans and index seeks any longer, these terms will be used interchangeably) and therefore that the overall aim of increasing query efficiency can be addressed by reducing this denominator to the lowest possible value and maximising $E(\sigma\varphi(R))$ towards $E(\sigma\varphi(R)) \rightarrow \text{LIM}(1)$. This means reducing the total available number of data values that must be read, by any technique, towards $|\sigma\varphi(R)|$. To do this, it is proposed to derive and implement new schema definitions in real-time to reach this goal, maximising the query efficiency.

To decide on the queries to analyse, the query cache can provide a short-term storage facility. Across RDBMS systems, the query cache is a local repository (normally held in memory while the RDBMS is active) of query statements, associated query execution plans and other metadata. The purpose of the cache is to minimise the time taken to generate query execution plans by re-using plans already generated [35]; this is appropriate for both exact query matches and queries which can be parameterised, i.e. literal values substituted with placeholders, such as with prepared queries. This cache can be repurposed; to analyse past query patterns and generate new supplementary schemata with appropriate cached query mappings. The proposed technique for doing this is presented in the next section.

There is precedence for mapping between sets and subsets within the relational model, defined in axiomatic set theory, for which set notation is used. The Zermelo-Fraenkel (ZF) axiom schema of separation [3] defines this, as shown in (19):

$$(\exists B)(\forall x)(x \in B \text{ iff. } x \in A \text{ and } \varphi(x)) \quad (19)$$

In straightforward terms, this means given the existence of a set B, for all members x in B, x is a member of B iff. x is a member of A and some predicate concerning x holds.

Translating this to the relational model it can be stated that given a subset schema B, for all rows of attribute values in B, those rows exist in B if and only if there exists a superset schema A and some predicate, or condition about those rows in B is true. Therefore, this is functionally equivalent to deriving a result set based on some predicate from a wider base schema, or in even simpler terms, equivalent to asserting that a query result is valid if it derives from a wider pool of available data. This axiom can be used to build subset schemas by analysing queries from the cache, deriving smaller subsets of attributes and predicates from those queries, and presenting these as materialised views (MVs) against which future iterations of the cached queries can be executed.

In RDBMS systems, views are overlays of relational expressions upon base schemas – essentially, query definitions which can be called using shorthand. An example follows - a view on table CUSTOMERS returns a subset of all attribute values from the Customer table bounded by the predicate ‘WHERE DateJoined > ’10 May 2019’ (20):

```
CREATE VIEW Customer AS
SELECT * FROM Customer WHERE DateJoined > '10 May 2019';      (20)
```

This query can also be defined using the relational algebra, as shown in (21):

$$\sigma_{(\text{DateJoined} > \text{'10 May 2019'})}(\text{Customer}) \quad (21)$$

Formally, this appears to be a simple implementation of the axiom schema of separation. However, views are illusory in the sense that although they provide a convenient shorthand to the user or calling application, when a query against a view is executed, the underlying definition of the view is called rather than any pre-prepared set of results. In other words, views alone do not provide any significant performance advantages over simply running the base queries; indeed, the sole advantage is human readability. For performance advantages, materialised views (MVs) are used for proof-of-concept implementation instead, which are persisted and the data within them stored separately to the base tables. Translating this idea back to the axiom schema of separation, the MV is set B, the base schema is set A and the predicates are the view definition.

The set of algorithms and the algorithmic implementation of this solution are presented in Chapter 8 and the accompanying Appendices.

5.8 Chapter Summary

This chapter introduced and described PETAS, a new multi-component system to supplement and augment the relational database query optimiser process within RDBMS engines.

Comprised of three key parts with various subcomponents, this chapter illustrated how the query parser can read an inbound SQL query and transform it into a compressed multidimensional matrix representation of itself, reflecting the key structures. It was shown how such multidimensional ‘cubes’ can be compared and contrasted through generating a third cube as a function of two cubes and using Hamming distances to calculate a similarity score, and shown how k-nearest neighbour can be used to identify previously-run queries similar to a query in hand, extracting the schema variant most likely to service a query effectively based on previous performance data. Finally, the dynamic schema redefinition process was discussed, a novel method of using the ZFC axiomatic schema of separation to define and destroy new schema subsets in real-time, implementable using techniques such as materialised views, and based upon usage and performance data from executed queries.

The following three chapters further enhance and explain each element of PETAS; the query parser (Chapter 6), the similarity scoring mechanism (Chapter 7) and the dynamic schema redefinition process (Chapter 8), and present the algorithms, implementations, tests, and results.

6.2 Design

This section expands upon the solution design for the query parser first described in Chapter 5.

SQL queries are multi-part, hierarchical objects with many properties and difficult to represent mathematically. To solve this problem, the query must be represented in some way that allows comparison against another one for similarity.

It is evident that a query can involve several relationships. Consider the query:

```
SELECT A.x, A.y, B.x FROM A INNER JOIN B ON A.z= B.z WHERE A.x = 10;
```

This query can be represented in the relational algebra like so:

$$\Pi_{A.x, A.y, B.x} (\sigma_{A.x=10} (A \theta B_{(A.z=B.z)}))$$

This query consists of a projection of columns x and y from table A, and column x from table B drawn from a selection with a predicate, based on two equi-joined relations. The predicate conditions are that columns z in A and B must be identical, and that column x from table A must be equal to 10. There is also an intersection of A on B, an equality relationship on $A.z = B.z$, membership of A by A.x, A.y, A.z, and membership of B by B.x, B.z (B.y is never specified).

These can, using the new proposed method, be alternatively modelled as a set of relationships with attributes. This means at a very basic level, a SQL query could be visualised as a kind of ‘query molecule’ based solely on these relationships (Fig. 6.2).

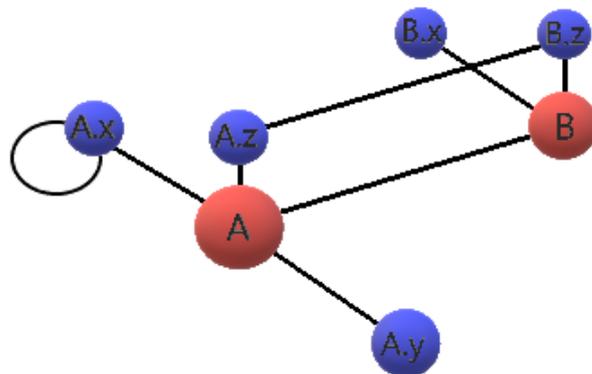


Fig. 6.2: Visualising a query in three dimensions

Note, this diagram shows only the relationships between entities (A, B) and attributes (A.x, A.y, A.z, B.x, B.z)), but this diagram is homomorphic to a directed graph. Directed graphs can be represented as matrices - particularly 'adjacency matrices'. These show, at a basic level, whether any two nodes in a graph are connected by an edge.

An adjacency matrix can be calculated for the query above. First, each node, and what node it is connected to (the direction), is listed as shown in Table 6.3.

Table 6.3: Example node relationship list

| Node From | Node To | Attribute Type |
|-----------|---------|----------------|
| A | A.x | PROJECTION |
| A | A.y | PROJECTION |
| A | A.z | MEMBER |
| A | B | INTERSECTION |
| A.x | A.x | PREDICATE |
| A.z | B.z | PREDICATE |
| B | A | INTERSECTION |
| B | B.z | MEMBER |
| B.z | A.z | PREDICATE |

From the node list, the adjacency matrix can be built, where 1 represents 'is a relationship', 0 represents 'no relationship'. The Y axis is 'Node From', the X axis 'Node To'. Note that there is no differentiation on attribute type at this stage, with the matrix having only two dimensions.

Table 6.4: Two-dimensional adjacency matrix

| | A | A.x | A.y | A.z | B | B.x | B.y | B.z |
|-----|---|-----|-----|-----|---|-----|-----|-----|
| A | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| A.x | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| A.y | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A.z | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| B | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| B.x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B.y | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B.z | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

This matrix shown in Table 6.4 is useful but doesn't take into account the type of relationship (edge). It simply measures, based on the fact that a relationship or edge exists. The type of relationship - hereafter called attribute type - can be a projection (SELECT), intersection (JOIN), member (e.g. A.x is a member of A) or predicate (either a JOIN predicate like A.z = B.z or a WHERE predicate like A.x = 10). These correspond to ZFC axiomatic set theory and the relational operations defined by Codd. This new method breaks new ground not through the definition of new set operations but by the representation of those operations in an adjacency matrix form.

This problem is approached in the same way as the simple representation above. The attribute type could be considered a dimension in its own right, on the Z-axis, as illustrated in Fig. 6.5:

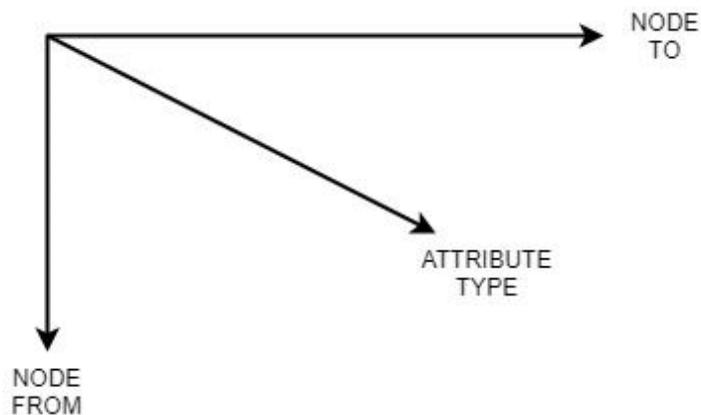


Fig. 6.5: Attribute type on the Z-axis.

This is difficult to represent in a single matrix, since it could be considered as multiple two-dimensional matrices - one matrix for each node from / node to set, for each attribute type. However, although it is difficult to visualise, it is not difficult to represent as a multidimensional array in code, and not difficult to conceptualise mathematically in Hilbert space [1] - Euclidean space extended from 2 to infinite dimensions (in this case, three).

Therefore, this two-dimensional matrix can be built up from the simple adjacency matrix for determining if there is an edge between two nodes to a more complex adjacency cube - adjacency matrices extended along the Z-axis - to determine if there is a relationship/edge between two nodes that is of a particular type (projection, intersection, member or predicate).

This allows the modelling of how similar two queries are at a lower level of granularity - i.e., is a relationship a SELECT (projection or member), a JOIN (intersection / predicate), or a WHERE (predicate)? This has some benefits: it reduces information loss in the translation from narrative SQL text to computational construction, and it is speculated that accuracy in similarity scoring will be improved through closer query matching.

Cubes are difficult to model in two-dimensional space, so one can visualise a three-dimensional adjacency cube laid out as 4 side-by-side matrices (4 slices of the cube on the Z axis) with the attribute type above it. Each attribute type (Join, Membership, Intersection or Predicate, indicated by their initials) as a slice of the cube on the Z axis - a layered representation. See Fig. 6.6 for this representation.

This approach can work for any number of dimensions in Hilbert space (although to visualise them in more than three dimensions requires some mental creativity), which means that any number of layered attribute properties could be included. This is useful because there are certain aspects of each query which are ignored in the test query above - such as predicate variable and value (A.x = 10 for example); whether a projected attribute (A.x) has any transforms upon it (e.g. CAST(A.x AS INT)) and what they are; dealing with INNER JOIN vs. OUTER JOIN; complex structures like Common Table Expressions; and subqueries or nested queries have yet to be modelled.

There are ways forward to include more detail in this process. The first is to model the adjacency matrices in a Hilbert space, an N-dimensional space where $N > 3$ and follow the same process. However, the cost of computation would very quickly rise as the number of dimensions increases. The degree of similarity between any two multidimensional cubes may also drop significantly since the ratio of overlapping data points to the available volume of the cube will reduce by an order of magnitude (since the volume of the cube has been increased by an order of magnitude).

| | J | | | | | | | | M | | | | | | | | I | | | | | | | | P | | | | | | | |
|-----|---|-----|-----|-----|---|-----|-----|-----|---|-----|-----|-----|---|-----|-----|-----|---|-----|-----|-----|---|-----|-----|-----|---|-----|-----|-----|---|-----|-----|-----|
| | A | A.x | A.y | A.z | B | B.x | B.y | B.z | A | A.x | A.y | A.z | B | B.x | B.y | B.z | A | A.x | A.y | A.z | B | B.x | B.y | B.z | A | A.x | A.y | A.z | B | B.x | B.y | B.z |
| A | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A.x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| A.y | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A.z | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| B | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B.x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B.y | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B.z | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

Fig. 6.6: 3D adjacency cube rendered in two dimensions

Another way to extend the model is to take inspiration from physics and consider real molecules. Molecules bind to each other, so complex structures like subqueries could be represented as a separate molecule with a bind on one or more nodes common to both. Thus, the adjacency matrices could be calculated separately, and the similarity scores combined in a different way (e.g., weighted mean). This direction is not pursued in the remainder of this research but discussed further in the conclusions later in this document.

6.2 Algorithmic Implementation

This process description can now be codified into an algorithm. The function should accept a database query written in SQL as input, and output an object representing the data in the expected format for an adjacency cube. To do this, a left-to-right parsing approach is taken as per Knuth [2] and others, similar to the approach used to build a parse tree in current database implementations (discussed in Chapter 3). However, rather than produce a parse tree, an internal map of relationships with relationship types is produced, represented as a multidimensional array; an object type supported by most, if not all, major programming languages.

Given an input database query, the following algorithm calculates the edges and types of edge, ready to be transmuted into a directed graph.

The algorithm begins by collating the elements of the query between the SELECT and FROM clauses. These are the projected elements of the query; those columns fetched from the database and displayed to the user. This is Algorithm 6.7.

Algorithm 6.7: Extracting projection elements from a SQL query

```
# Extract projection elements to list 'edges'
-----
define list 'edges' as empty list
define variable 'node' as empty untyped variable
define variable 'SELECTs' as empty string variable
define variable sqlQueryA as the input SQL query, rendered as a string
set SELECTs = substring of sqlQueryA between position 6 and up to
  but not including first instance of word 'FROM'
set SELECTs = SELECTs, trimmed of whitespace
# loop
do while length of SELECTs > 0:
  --if first char in SELECTs != ",":
  ----set node = node + first char in SELECTs
  ----set SELECTs = SELECTs from 2nd char to end
  --else:
  ----set node = node, trimmed of whitespace
```

```

----to an unnamed 3-item list, append var 'node' from the first position
----  to the first instance of '.'; node; and the string 'PROJECTION'
----append this list as an element in list 'edges'
----set node to an empty untyped string
----set SELECTs = SELECTs from 2nd char to end
set node = node, trimmed of whitespace
to an unnamed 3-item list, append var 'node' from the first position
  to the first instance of '.'; node; and the string 'PROJECTION'
append this list as an element in list 'edges'

```

The input to Algorithm 6.7 is the SQL query, designated sqlQueryA.

Algorithm 6.7 simply loops through each object SELECTed in the query, identifies the object as a projection upon the remainder of the query and outputs these facts to a multidimensional [1,3] list or array.

The output of this process is a list with dimensions [1,3], each row representing a column or database object that is SELECTed from the query; in the test query, this is A.x, A.y and B.x.

Next, the contents of the input query are parsed to extract the relations, or tables, from which these projections are taken – the table names. These tables intersect in zero or more ways (no join conditions mean no intersections other than on itself; several tables imply several intersections) so each element is marked as an intersection in the output array. This proceeds as follows in Algorithm 6.8.

Algorithm 6.8: Extracting membership elements from a SQL query

```

# Extract FROM clause elements
define variable 'FROMs' as a substring of sqlQueryA between the first instances
  of 'FROM and 'WHERE'
set FROMs = FROMs, trimmed of whitespace
define variable 'nodeFrom' as an empty untyped variable
define variable 'nodeTo' as an empty untyped variable
define variable 'entities' and set as a substring of FROMs from first character
  to first occurrence of 'ON' or first occurrence of WHERE or end of string.
set nodeFrom as substring of entities from first char to first whitespace (first word)
set nodeTo as substring of entities from first occurrence of JOIN to end of string
set nodeTo = nodeTo, trimmed of whitespace
define variable nodeFromEntity set to value of nodeFrom (value assignment)
define variable nodeToEntity set to value of nodeTo (value assignment)
to an unnamed 3-item list, append nodeFrom, nodeTo and 'INTERSECTION'
append this list as an element in list 'edges'

#parse the JOIN predicates
define variable 'PREDICATES' as an empty untyped variable
set PREDICATES = substring of FROMs from first occurrence of ON to end of string,
  trimmed of whitespace
set nodeFrom = substring of PREDICATES from first character to first occurrence of
  whitespace

```

```

set PREDICATES = substring of PREDICATES from first occurrence of whitespace to end of
string,
    trimmed of whitespace
set nodeTo = substring of PREDICATES from first occurrence of whitespace to end of string
to an unnamed 3-item list, append nodeFromEntity, nodeFrom and 'MEMBERSHIP'
append this list as an element in list 'edges'
to an unnamed 3-item list, append nodeToEntity, nodeFrom and 'MEMBERSHIP'
append this list as an element in list 'edges'

# repeat both code blocks above, adjusting input variable sqlQueryA
or var 'FROMs', for multiple JOIN predicates.

```

The input to Algorithm 6.8 is sqlQueryA (therefore Algorithms 6.7 and 6.8 could be run in parallel). This algorithm extracts the JOIN clauses from the query and identifies the source (left) and destination (right) for each JOIN. Predicates are not yet considered. This is done through word-by-word parsing of the portion of the SQL query between the clause indicators FROM and WHERE and is repeatable for multiple JOIN predicates.

The output of this algorithm is a [1,3]-shaped multidimensional array or list containing ordered tuples of the source, destination and 'INTERSECTION' string for each intersection identified in the query.

Algorithm 6.9: Extracting predicates from a SQL query

```

# parse the WHERE clause
define variable 'WHEREs' as an empty untyped variable
set WHEREs = substring of sqlQueryA from first occurrence of 'WHERE' to end of string,
trimmed of whitespace
replace all semicolons in WHEREs with empty strings
define new variable 'andFlag' as Boolean-typed variable initialised to 0
define new variable 'orFlag' as Boolean-typed variable initialised to 0
if 'AND' in WHEREs:
--set andFlag = 1
if 'OR' in WHEREs:
--set orFlag = 1

# further variable declarations
define variable 'leftSide' as an empty untyped variable
define variable 'rightSide' as an empty untyped variable
define variable 'nodeFrom' as an empty untyped variable
define variable 'nodeTo' as an empty untyped variable

# parse the WHERE clause in the case that it does contain Boolean expressions AND or OR
while exists 'AND' or 'OR' in WHEREs variable, do:
--while andFlag = 1
----set leftSide = substring of WHEREs from first character to first whitespace
----# check for primitives and set the right-hand side of the predicate accordingly
----if WHEREs contains the string '=':
-----set rightSide = substring of WHEREs from the first occurrence of '=' to the first
----- occurrence of 'AND'
----if WHEREs contains the string '>':
-----set rightSide = substring of WHEREs from the first occurrence of '>' to the first
----- occurrence of 'AND'
----if WHEREs contains the string '<':

```

```

-----set rightSide = substring of WHEREs from the first occurrence of '<' to the first
----- occurrence of 'AND'
----if WHEREs contains the string '<>':
-----set rightSide = substring of WHEREs from the first occurrence of '<>' to the first
----- occurrence of 'AND'
----# note this list can be extended to all legal primitives in SQL, including IN/LIKE,
---- >=, <=, IS, CONTAINS, EXISTS and so on
----# now assess to see if the right side is a column or a literal
----set nodeFrom = leftSide
----if rightSide can be converted without error to a real number
---- or rightSide contains a single quote ':
-----set nodeTo = nodeFrom
----else if rightSide contains a full stop # (looks like a column name)
-----set nodeTo = rightSide
----to an unnamed 3-item list, append nodeFrom, nodeTo and 'PREDICATE'
----append this list as an element in list 'edges'
----if the string 'AND' exists in WHEREs:
-----set WHEREs = substring of WHEREs from the first occurrence of AND
----- to the end of string, trimmed of whitespace
----set andFlag = 1
----else:
-----set andFlag = 0
----# end of inner loop

--# now parse the OR statements, if they exist, from the WHERE clause
-- (follows same pattern as ANDs)
--while orFlag = 1
----set leftSide = substring of WHEREs from first character to first whitespace
----# check for primitives and set the right-hand side of the predicate accordingly
----if WHEREs contains the string '=':
----if WHEREs contains the string '>':
-----set rightSide = substring of WHEREs from the first occurrence of '=' to the first
----- occurrence of 'OR'
----if WHEREs contains the string '>':
-----set rightSide = substring of WHEREs from the first occurrence of '>' to the first
----- occurrence of 'OR'
----if WHEREs contains the string '<':
-----set rightSide = substring of WHEREs from the first occurrence of '<' to the first
----- occurrence of 'OR'
----if WHEREs contains the string '<>':
-----set rightSide = substring of WHEREs from the first occurrence of '<>' to the first
----- occurrence of 'OR'
----# note this list can be extended to all legal primitives in SQL, including IN/LIKE,
---- >=, <=, IS, CONTAINS, EXISTS and so on
----# now assess to see if the right side is a column or a literal
----set nodeFrom = leftSide
----if rightSide can be converted without error to a real number
---- or rightSide contains a single quote ':
-----set nodeTo = nodeFrom
----else if rightSide contains a full stop # (looks like a column name)
-----set nodeTo = rightSide
----to an unnamed 3-item list, append nodeFrom, nodeTo and 'PREDICATE'
----append this list as an element in list 'edges'
----if the string 'OR' exists in WHEREs:
-----set WHEREs = substring of WHEREs from the first occurrence of OR
----- to the end of string, trimmed of whitespace
----set orFlag = 1
----else:
-----set orFlag = 0
----# end of inner loop
# end of loop

# in the case of a simple WHERE clause with no additional predicates

```

```

set nodeFrom = substring of WHEREs from first char to first occurrence of whitespace,
  trimmed of whitespace
set WHEREs = substring of WHEREs from first occurrence of whitespace to end of string,
  trimmed of whitespace
if rightSide can be converted without error to a real number
  or rightSide contains a single quote ':
--set nodeTo = nodeFrom
else if rightSide contains a full stop # (looks like a column name)
--set nodeTo = WHEREs
to an unnamed 3-item list, append nodeFrom, nodeTo and 'PREDICATE'
append this list as an element in list 'edges'

```

Algorithm 6.9 parses the WHERE clause. It first looks for the condition that AND or OR exist in the predicate list within the query (after the WHERE clause) indicating multiple predicates to be parsed; if this is the case, the predicates are extracted separately, the left-hand and right-hand elements of each predicate pair are extracted according to the primitive used, and a [1,3] multidimensional array or list captures output as left-side (nodeFrom), right-side (nodeTo) and 'PREDICATE'. In the case that a single predicate is used (no AND or OR statement is used), the same process is used but once only. Algorithm 6.9 outputs the list 'edges' as a [1,3] multidimensional array and thus can be run in parallel alongside Algorithms 1 and 2 if the 'edges' arrays are concatenated afterwards.

The output array 'edges' is now used to build the adjacency cube, assuming Algorithms 6.7 - 6.9 have been processed. To do this a series of 2-dimensional lists are built, one for each attribute type, then inserted into a 3-dimensional list using simple if-then control flow logic. It would be better for performance to execute this operation in SQL as it has set support i.e. parallelism rather than iterating through each list, however the method is irrespective providing the outcome is the same and OOP languages have threading which implies parallelism could be used in the iterative method.

Algorithm 6.10: Function variant of query parser

```

function buildEdgeArray (sqlQueryA):
  <Algorithm 1>
  <Algorithm 2>
  <Algorithm 3>
return edges

```

Algorithm 6.10 wraps the edge list generation code (Algorithms 6.7 – 6.9) in a simple function, taking sqlQueryA as input and outputting a single multidimensional array, 'edges', size [n,3].

The next step is to convert this edges array to an adjacency cube. Algorithm 6.11 describes this process, which takes an edge list as input, sorts and deduplicates the list, iterates through the list, identifies all pairs of relationships per attribute type and appends these to an output list, 'cube'.

Algorithm 6.11: Converting an edge list to an adjacency cube

```
Sort the edge list
Deduplicate the edge list
For each type of attribute JOIN, INTERSECTION, MEMBERSHIP, PREDICATE:
  For each nodeFrom/nodeTo (data point in the current attribute type slice):
    If a relationship exists in edges for the current attribute type, mark with a
    1
    Else mark with a 0
Once complete, append an [n, 4] list to 'cube', a new multidimensional list:
[nodeFrom, nodeTo, attributeType, value (0 or 1)]
```

The next step is to use these cubes as input to the similarity scoring function. This is discussed further in Chapter 8.

6.3 Practical Implementation

The following Code Listings 6.12 – 6.15 show the implementation of Algorithms 6.7 – 6.10 in section 6.2. These were implemented in Python 3.

Code Listing 6.12: Algorithm 6.7 in Python

```
# Extract WHERE clause elements
SELECTs = sqlQueryA[sqlQueryA.find("SELECT")+6:sqlQueryA.find("FROM")].strip();
edges = [];
node = "";

while len(SELECTs) > 0:
  if SELECTs[0:1] != ",":
    node = node + SELECTs[0:1];
    SELECTs = SELECTs[1:];
  else:
    node = node.strip();
    edges.append([node[:node.find(".")], node, "PROJECTION"]);
    node = "";
    SELECTs = SELECTs[1:];
node = node.strip();
edges.append([node[:node.find(".")], node, "PROJECTION"]);
```

Code Listing 6.13: Algorithm 6.8 in Python

```
# Extract FROM clause elements
FROMs = sqlQueryA[sqlQueryA.find("FROM")+4:sqlQueryA.find("WHERE")].strip();
nodeFrom = "";
nodeTo = "";
entities = FROMs[0:FROMs.find("ON")].strip();
```

```

nodeFrom = entities[0:entities.find(" ")]
nodeTo = entities[entities.find("JOIN")+4:].strip()
nodeFromEntity = nodeFrom
nodeToEntity = nodeTo
edges.append([nodeFrom, nodeTo, "INTERSECTION"])
edges.append([nodeTo, nodeFrom, "INTERSECTION"])

# deal with JOIN predicates (repeatable)
PREDICATES = FROMs[FROMs.find("ON")+2:].strip()
nodeFrom = PREDICATES[0:PREDICATES.find(" ")]
PREDICATES = PREDICATES[PREDICATES.find(" ")+1:].strip()
nodeTo = PREDICATES[PREDICATES.find(" ")+1:]
# add the memberships
edges.append([nodeFromEntity, nodeFrom, "MEMBER"])
edges.append([nodeToEntity, nodeTo, "MEMBER"])
# add the JOIN predicate
edges.append([nodeFrom, nodeTo, "PREDICATE"])

```

Code Listing 6.14: Algorithm 6.9 in Python

```

# deal with WHERE clause
WHEREs = sqlQueryA[sqlQueryA.find("WHERE")+5:].strip()
WHEREs = WHEREs.replace(";", "");
andFlag = 0;
orFlag = 0;
# deal with multiple clauses (only AND and OR supported)
if "AND" in WHEREs:
    andFlag = 1;
if "OR" in WHEREs:
    orFlag = 1;

while "AND" in WHEREs or "OR" in WHEREs:
    while andFlag == 1:
        leftSide = WHEREs[0:WHEREs.find(" ")]
        if "=" in WHEREs:
            rightSide = WHEREs[WHEREs.find("=")+1:WHEREs.find("AND")].strip()
        if ">" in WHEREs:
            rightSide = WHEREs[WHEREs.find(">")+1:WHEREs.find("AND")].strip()
        if "<" in WHEREs:
            rightSide = WHEREs[WHEREs.find("<")+1:WHEREs.find("AND")].strip()
        if "<>" in WHEREs:
            rightSide = WHEREs[WHEREs.find("<>")+2:WHEREs.find("AND")].strip()
        # Assess to see if right side is a column or a literal
        nodeFrom = leftSide
        if rightSide.isdigit() == True: # is a number
            nodeTo = nodeFrom
        elif "'" in rightSide: # is a string
            nodeTo = nodeFrom
        elif "." in rightSide: # looks like a column name
            nodeTo = rightSide
        edges.append([nodeFrom, nodeTo, "PREDICATE"])
        if "AND" in WHEREs:
            WHEREs = WHEREs[WHEREs.find("AND")+3:].strip()
            andFlag = 1;
        else:
            andFlag = 0;

    if "OR" in WHEREs:
        orFlag = 1;
    while orFlag == 1:
        leftSide = WHEREs[0:WHEREs.find(" ")]

```

```

if "=" in WHEREs:
    rightSide = WHEREs[WHEREs.find("=")+1:WHEREs.find("OR")].strip();
if ">" in WHEREs:
    rightSide = WHEREs[WHEREs.find(">")+1:WHEREs.find("OR")].strip();
if "<" in WHEREs:
    rightSide = WHEREs[WHEREs.find("<")+1:WHEREs.find("OR")].strip();
if "<>" in WHEREs:
    rightSide = WHEREs[WHEREs.find("<>")+2:WHEREs.find("OR")].strip();
# Assess to see if right side is a column or a literal
nodeFrom = leftSide;
if rightSide.isdigit() == True: # is a number
    nodeTo = nodeFrom;
elif "'" in rightSide: # is a string
    nodeTo = nodeFrom;
elif "." in rightSide: # looks like a column name
    nodeTo = rightSide;
edges.append([nodeFrom, nodeTo, "PREDICATE"]);
if "OR" in WHEREs:
    WHEREs = WHEREs[WHEREs.find("OR")+2:].strip();
    orFlag = 1;
else:
    orFlag = 0;

# no ANDs or ORs, simple single WHERE
nodeFrom = WHEREs[:WHEREs.find(" ")].strip();
WHEREs = WHEREs[WHEREs.find(" ") + 3:].strip();
if WHEREs.isdigit() == True:
    nodeTo = nodeFrom;
elif "'" in WHEREs: # is a string
    nodeTo = nodeFrom;
elif "." in WHEREs: # looks like a column name
    nodeTo = WHEREs;
edges.append([nodeFrom, nodeTo, "PREDICATE"]);

```

Code Listing 6.15: Algorithm 6.10 in Python

```

def buildEdgeArray (sqlQueryA):
    <Code Listing 1>
    <Code Listing 2>
    <Code Listing 3>
    return edges;

```

Fig. 6.16 shows working example output from Code Listing 6.15 using the test query from the beginning of this chapter.

```

Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
Edges for SELECT A.x, A.y, B.x FROM A INNER JOIN B ON A.z = B.z
WHERE A.x = 10;

['A', 'A.x', 'PROJECTION']
['A', 'A.y', 'PROJECTION']
['B', 'B.x', 'PROJECTION']
['A', 'B', 'INTERSECTION']
['B', 'A', 'INTERSECTION']
['A', 'A.z', 'MEMBER']
['B', 'B.z', 'MEMBER']
['A.z', 'B.z', 'PREDICATE']
['A.x', 'A.x', 'PREDICATE']

```

Fig. 6.16: Example screenshot from edge list builder

Code Listing 6.17 shows the working implementation of Algorithm 6.11.

Code Listing 6.17: Conversion to multidimensional array

```

# deduplicate edges
foo = [];
for i in edges:
    if i not in foo:
        foo.append(i);
edges = foo;

# get distinct list of nodes from edges
distinctNodes = [];
counter = 0;

for i in edges:
    if i[0] not in distinctNodes:
        distinctNodes.append(i[0]);
    if i[1] not in distinctNodes:
        distinctNodes.append(i[1]);

distinctNodes.sort();

projection = [];
intersection = [];
member = [];
predicate = [];
appendFlag = 0;

# build the projection list:
for i in distinctNodes:
    for j in distinctNodes:
        for k in range (0, len(edges)):
            if i == edges[k][0] and j == edges[k][1] and edges[k][2] == "PROJECTION":
                projection.append([i,j,1]);
                appendFlag = 1;
        if appendFlag == 0:
            projection.append([i,j,0]);

```

```

appendFlag = 0;

# build the intersection list
for i in distinctNodes:
    for j in distinctNodes:
        for k in range (0, len(edges)):
            if i == edges[k][0] and j == edges[k][1] and edges[k][2] == "INTERSECTION":
                intersection.append([i,j,1]);
                appendFlag = 1;
            if appendFlag == 0:
                intersection.append([i,j,0]);
            appendFlag = 0;

# build the member list
for i in distinctNodes:
    for j in distinctNodes:
        for k in range (0, len(edges)):
            if i == edges[k][0] and j == edges[k][1] and edges[k][2] == "MEMBER":
                member.append([i,j,1]);
                appendFlag = 1;
            if appendFlag == 0:
                member.append([i,j,0]);
            appendFlag = 0;

# build the predicate list
for i in distinctNodes:
    for j in distinctNodes:
        for k in range (0, len(edges)):
            if i == edges[k][0] and j == edges[k][1] and edges[k][2] == "PREDICATE":
                predicate.append([i,j,1]);
                appendFlag = 1;
            if appendFlag == 0:
                predicate.append([i,j,0]);
            appendFlag = 0;

# merge the lists into a cube
cube = [];
cube.append(projection);
cube.append(intersection);
cube.append(member);
cube.append(predicate);

```

Code Listing 6.17 can then be wrapped in a function, as shown in Code Listing 6.18:

Code Listing 6.18: Functionalised adjacency cube build code

```

def buildAdjacencyCube(edges):
    <Code Listing 6.17>
    return cube;

```

This enables the process to start with an input, `sqlQueryA`, turn it into an edge list, and transform the edge list into an adjacency cube through two function calls.

Note that ‘cube’ is a nested list.

- Level 0: Contains 4 lists, one for each NodeFrom/NodeTo/Value tuple.
- Level 1: Contains a NodeFrom/NodeTo/Value tuple.

To refer to or query a particular element (Python is zero-indexed), as an example:

```
> print(cubeA[2][8])  
  
['A.x', 'A', 0]
```

There are 64 (square of 8 distinct nodes) lists in Level 1 multiplied by 4 attribute types in level 0, equalling 256 values in the adjacency cube for the test query.

6.4 Experimental Design

Microsoft SQL Server was used to write a SQL script to generate 1,000 SQL queries, ranging in complexity. For each of the queries, the code was augmented with exception handling and performed two sets of tests; the first, to establish the proportion of queries for which the implementation is able to parse without error, a simple statistical count; and the second, the duration of the process in milliseconds, to assess how much overhead the process might place on an RDBMS if implemented as an augmentation.

The testing in this area is tightly bound with the experimental design and testing presented in Chapter 7 (similarity scoring). The algorithms are used from this chapter in the larger round of testing against real-life data in Chapter 7, and to that end the random query generator was employed for this set of tests suitable for further use against the real-life data examined later, namely Chicago crime data by geographic region in a limited range of years. More information on this data set is presented in Chapter 7.

The code listing for the SQL-based random query generator is shown in Code Listing 6.19.

Code Listing 6.19: Random query generator for Chicago crime data

```
SET NOCOUNT ON  
GO  
  
DROP PROCEDURE IF EXISTS dbo.chicagoQueryGenerator  
GO  
  
CREATE PROCEDURE dbo.chicagoQueryGenerator  
AS BEGIN  
  
DECLARE @columnCount TINYINT  
DECLARE @counter TINYINT = 0
```

```

DECLARE @thisColumn VARCHAR(255)
DECLARE @select VARCHAR(500) = 'SELECT '
DECLARE @used TABLE ( [name] VARCHAR(255) )

SET          @columnCount = CEILING((
                SELECT      TOP 1 c.[column_id]
                FROM          sys.columns c
                INNER        JOIN sys.tables t ON c.object_id = t.object_id
                WHERE        t.[name] = 'chicagobase'
                ORDER        BY NEWID() ) / 2.0)

WHILE @counter < @columnCount
BEGIN
    SET @thisColumn = (
        SELECT      TOP 1 c.[name]
        FROM          sys.columns c
        INNER        JOIN sys.tables t ON c.object_id = t.object_id
        LEFT         JOIN @used u ON c.[name] = u.[name]
        WHERE        u.[name] IS NULL
        AND          t.[name] = 'chicagobase'
        ORDER BY NEWID() )
    INSERT INTO @used VALUES ( @thisColumn )
    SET @select = @select + @thisColumn + ', '
    SET @counter += 1
END
SET @select = LEFT(@select, LEN(@select) - 1) + ' '

DECLARE @from VARCHAR(500) = ' FROM chicagoBase' + ' '

DECLARE @where VARCHAR(500) = 'WHERE (1=1)' + ' '
-- pick a random number of where clauses, between 0 and 2
DECLARE @numOfWheres TINYINT = ( SELECT ABS(CHECKSUM(NEWID())) % 3 )
DECLARE @colName VARCHAR(255), @dType VARCHAR(255), @val VARCHAR(255)
DECLARE @operator TINYINT, @letters TINYINT
WHILE @numOfWheres > 0
BEGIN
    -- pick a random column from the chicagoBase table
    SELECT @colName = c.[name], @dType = y.[name]
    FROM      sys.columns c
    INNER    JOIN sys.types y ON c.system_type_id = y.system_type_id
    WHERE    c.object_id = OBJECT_ID('chicagoBase')
    AND      c.column_id = ( SELECT ABS(CHECKSUM(NEWID())) %
                            ( SELECT COUNT(*) FROM sys.columns c
                              WHERE c.object_id = OBJECT_ID('chicagoBase') ) + 1 )
    )

    -- now select a random value corresponding to the datatype of
    -- the randomly chosen column
    IF @dType = 'bit' SET @val = CAST(ABS(CHECKSUM(NEWID())) % 2) AS
VARCHAR(255))
    IF @dType LIKE ('%tinyint%')
        SET @val = CAST(ABS(CHECKSUM(NEWID())) % 255) AS VARCHAR(255))
    IF @dType = 'datetime'
        SET @val = '''' + CONVERT(VARCHAR,
DATEADD(MINUTE, (ABS(CHECKSUM(NEWID()))
% 2629800) * -1, GETDATE()), 120) + '''' -- any time in last 5 years
    IF @dType IN ('decimal', 'numeric', 'float') SET @val =
CAST((ABS(CHECKSUM(NEWID())) % 5000) +
((ABS(CHECKSUM(NEWID())) % 100)/100.0) AS VARCHAR(255))
    IF @dType IN ('varchar') BEGIN
        SET @val = ''
        SET @letters = ABS(CHECKSUM(NEWID())) % 10 + 1
        WHILE @letters > 0 BEGIN
            SET @val = @val + CHAR(ABS(CHECKSUM(NEWID())) % 26 + 96)

```

```

-- up to 10 random lowercase ASCII characters
SET @letters -= 1

END
SET @val = '' + @val + ''
END

-- construct the WHEREs
SET @operator = ABS(CHECKSUM(NEWID())) % 4 + 1
SET @where = @where + 'AND ' + @colName + ' ' +
CASE WHEN @operator = 1 THEN '='
      WHEN @operator = 2 AND @val NOT LIKE ('%' + @val + '%') THEN '>'
      WHEN @operator = 2 AND @val LIKE ('%' + @val + '%') THEN '='
      WHEN @operator = 3 AND @val NOT LIKE ('%' + @val + '%') THEN '<'
      WHEN @operator = 3 AND @val LIKE ('%' + @val + '%') THEN '='
      WHEN @operator = 4 THEN '!=' END
SET @where = @where + ' ' + @val + ' '
SET @numOfWherees -= 1
END

-- remove the WHERE (1=1) placeholder
IF @where NOT LIKE ('% AND %')
SET @where = REPLACE(@where, 'WHERE (1=1) ', '')
ELSE
SET @where = REPLACE(@where, 'WHERE (1=1) AND', 'WHERE')

-- concatenate into a statement
DECLARE @output VARCHAR(1000) = @select + @from + ISNULL(@where, '') + ';'
SELECT @output
END
GO

```

Fig. 6.20 shows a screenshot illustrating some of the generated queries:

| rid | stmt |
|-----|--|
| 1 | SELECT rCommunityArea, rcaseNumber FROM chicagoBase ; |
| 2 | SELECT rWard, rcaseNumber, rLatitude, rUpdatedOn, rDistrict, rCommunityArea, rDate, r |
| 3 | SELECT ryCoordinate, rArrest, rLocation, rCommunityArea, rYear, rDate FROM chicagoBa |
| 4 | SELECT rDescription, rBeat, rLongitude FROM chicagoBase ; |
| 5 | SELECT rcaseNumber, rIUCR, rDistrict, rBeat, rDescription, rid FROM chicagoBase ; |
| 6 | SELECT rIUCR, rBeat, rPrimaryType, rDate FROM chicagoBase ; |
| 7 | SELECT rDate, rDomestic, rDescription, rDistrict, rLocation, rFBICode FROM chicagoBase |
| 8 | SELECT rid, rBeat, rCommunityArea, rPrimaryType, rFBICode FROM chicagoBase ; |
| 9 | SELECT rDate FROM chicagoBase WHERE rDate = '2014-10-06 12:00:24'; |
| 10 | SELECT rWard, ryCoordinate, rIUCR, rBlock, rxCoordinate, rDistrict, rBeat FROM chicagoE |
| 11 | SELECT rLocation, rDate, rid, rBeat, rLocationDescription FROM chicagoBase ; |
| 12 | SELECT rFBICode, rArrest, ryCoordinate, rPrimaryType, rid, rBeat, rxCoordinate, rDescrip |
| 13 | SELECT rWard, rYear, rcaseNumber, rxCoordinate, rid, rPrimaryType, rLongitude, ryCoord |
| 14 | SELECT rDomestic, rBlock FROM chicagoBase ; |
| 15 | SELECT rPrimaryType, rxCoordinate, rid, rDescription, rLocation, rIUCR, rLongitude, rBea |
| 16 | SELECT rBeat, rLocationDescription, rIUCR, rLatitude, rcaseNumber FROM chicagoBase ; |
| 17 | SELECT rPrimaryType, rcaseNumber, rxCoordinate, rWard, rYear, rDescription, rLocation |

Fig. 6.20: Randomly generated queries against the Chicago data set

6.5 Testing and Results

First, 1,000 random queries was created using the generator against the Chicago crime data set, about which more details are provided in the next chapter. Of these 1,000 queries, 947 were valid; 53 were manually removed with parsing errors (success rate of 94.7%).

Each query in this set was then executed against the adjacency cube generator, noting the presence or not of an exception after the edge list generator, and after the cube generator, respectively. The duration of each conversion from query to adjacency cube was measured in milliseconds. Table 6.21 and Fig. 6.22 below illustrate the findings.

Table 6.21: Functional testing of query to cube transformation

| | |
|--|--------|
| Random queries successfully generated | 947 |
| Random queries unsuccessfully generated | 53 |
| Success rate | 94.7% |
| Edge lists successfully generated | 947 |
| Edge lists unsuccessfully generated | 0 |
| Success rate | 100.0% |
| Adjacency cubes successfully generated | 947 |
| Adjacency cubes unsuccessfully generated | 0 |
| Success rate | 100.0% |

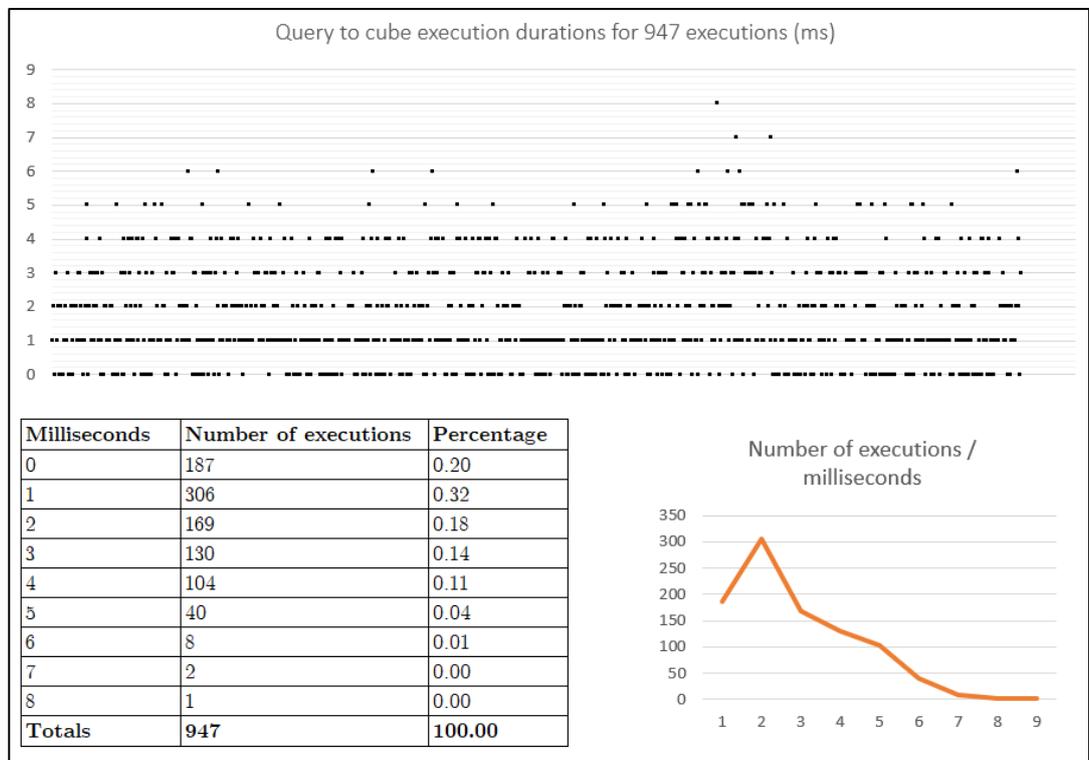


Fig. 6.22: Duration statistics for query to cube transformation

Of the 947 queries randomly generated, 100% were successful in edge generation and 100% were subsequently successful in cube generation.

Durations ranged from 0 to 9ms in whole increments (microsecond duration granularity was unavailable in the test harness): the mean average duration was 1.8ms with 1.5ms standard deviation. Approximately 634 executions, or 67%, completed within 1 standard deviation of the mean duration.

Through examination of the output artefacts, it was noted that in some cases objects within queries were mis-parsed. Most commonly, this occurred when end-of-line characters were encountered; where multiple JOIN conditions were specified in different syntax to that expected; and so on. This led to duplicate edges listed in some output artefacts, or edges listed minus the first or last character of their names. Inconsistent production of MEMBERSHIPs was also noted in the slice. Through trial and error, most of these issues were virtually eliminated for the next round of testing, presented in Chapter 7, although more work is required to extend this parser to the whole range of SQL syntax.

6.6 Conclusions

A series of short algorithms were constructed to parse SQL database queries into subsets of projections, intersections, memberships, and predicates. In general, the construction of the algorithms, the transition from algorithms to implementations, and the testing was successful; generation of adjacency cubes was successful and typically took place in an average of less than 2ms per query, well within the expected runtime of a query within an RDBMS; the principles of transitioning from a narrative object such as a SQL query to a comparable and computable form, the multidimensional array, were demonstrated; and the algorithms were demonstrated to be fairly robust.

However, there were some issues encountered, particularly in parsing. Parsers, as a superset of database query parsers, have a rich and detailed history in the research literature, with giants in the field such as Donald Knuth [2] devoting considerable years to their construction and correct implementation. It is unlikely that a perfect query parser could be recreated within the confines of a single research project, and to that extent the artefact does not have the full range of SQL support that would be ideal; for example, it will not support common table expressions; non-primitive comparison operators (IN, LIKE, BETWEEN); non-standard JOIN types, OUTER or CROSS APPLY operations, or JOINS with AND or OR conditions (these can be specified in the WHERE clause). Occasional parsing errors in object names were observed, and in some cases

MEMBERSHIPs in particular were not generated correctly, although it was validated and verified that all other attribute types were generated successfully. Additional whitespace, end-of-line characters and other terminators also frustrated the parser.

For the benefit of better quality, in the future, the use of an industry-standard parser is proposed which benefits from the long tail of research and development from the community, such as GNU Bison [3] (used in MySQL). The benefits of such a parser were discussed in Chapter 3, section 3.5.3. The outputs of such a parser, the parse tree, could be used as inputs for the adjacency cube generator.

6.7 Chapter Summary

In this chapter, the solution design for the query parser was extended, incorporating the adjacency cube generation mechanism, from the theoretical solution described in Chapter 5 to a set of algorithms. The experimental approach to testing was outlined, generating 1,000 realistic queries against a data set and applying the algorithmic designs to a set of implemented scripts in Python. The scripts were tested against these generated queries and the rate of exceptions (0%) and the duration of the process (average 1.8ms per execution) were recorded. Deficiencies were also observed in the implementation that are solvable either by developing the parser further to be more robust or incorporating an industry-standard parser which has the benefit of many years of research and development. In doing so, adjacency cubes could be generated directly from the parse tree output.

In the next chapter, the solution is extended into the similarity scoring mechanism and the test data set is further introduced. The algorithms and code listings are incorporated from this chapter with extended algorithms and code for K-nearest-neighbour driven similarity scoring to present an implementation of the end-to-end real-time mechanism of PETAS. The performance and outputs are tested, and the outcomes of the tests are presented.

Chapter 7 – Testing: Similarity Scoring and Schema Selection

7.1 Introduction

In this chapter, the ability to successfully generate adjacency cubes from input SQL database queries as described in Chapter 6 is assumed, and the similarity scoring mechanism and the schema selector as described are implemented and tested for the next part of the solution.

As recalled from the solution design, the next steps between the generation of an adjacency cube and handing off a new database query to the ordinary query engine are the execution of the scoring mechanism to generate a score based on the similarity or otherwise between two input adjacency cubes; the KNN selector mechanism, which takes as input pairs of adjacency cubes and clusters similar adjacency matrices according to score distance; the schema classifier, which takes as input the adjacency cube for the query at hand and selects, using the KNN selector, the appropriate schema for it based on successful schema classifications of prior queries, according to score distance; and finally the query mapper, which works independently to adjust the query at hand to fit the recommended schema to ensure both syntactic and functional validity.

7.1.1 Similarity scoring

First, similarity scoring is addressed. As described in Chapter 5, a relative score can be calculated between two adjacency cubes A and B, which consist of X-Y-Z intersections, each marked with 0 or 1 depending on whether a relationship exists between the node-node-attribute type of the tuple, by comparison of the structure of each query. This structure does not take into consideration the actual objects; the cube A, projecting 5 columns from relation R with no joins or predicates, would look structurally similar to cube B, projecting a different 5 columns from relation R. However, from a structural perspective these two cubes are similar and consequently would merit a high similarity score. The strictly structural approach needs augmentation with a method that compares the objects within the query, adjusting the score accordingly.

First, two example queries are considered, Q1 and Q2. These queries are listed below. The adjacency cube transformation is used to turn these two queries from SQL to edge list to cube. This process is shown in Tables 7.1 and 7.2 below, and in Figs. 7.3 and Fig. 7.4.

for each intersection of C1 and C2, calculation of the appropriate C3 result is done using the following formula (1), which simply subtracts one value from the other at each intersection and squares the result (which has the effect of applying an absolute function to the output, eliminating negative 1):

$$C_{3i,j,k} = (C_{2i,j,k} - C_{1i,j,k})^2, \forall([i, j, k]) \quad (1)$$

Thus, Equation (1) is applied to the two adjacency cubes which renders the following resulting adjacency cube (Fig. 7.5):

| | J | | | | | | | | M | | | | | | | | I | | | | | | | | P | | | | | | | | |
|-----|---|-----|-----|-----|---|-----|-----|-----|---|-----|-----|-----|---|-----|-----|-----|---|-----|-----|-----|---|-----|-----|-----|---|-----|-----|-----|---|-----|-----|-----|---|
| | A | A.x | A.y | A.z | B | B.x | B.y | B.z | A | A.x | A.y | A.z | B | B.x | B.y | B.z | A | A.x | A.y | A.z | B | B.x | B.y | B.z | A | A.x | A.y | A.z | B | B.x | B.y | B.z | |
| A | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A.x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A.y | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A.z | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B.x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B.y | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B.z | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

Fig. 7.5: Resulting adjacency cube C3

Equation (2) is now applied to calculate the similarity score – the sum of all 1s in C3 is 10, divided by 2 is 5 (the numerator in (2)); the cardinality of the edge list is 9 (the denominator in (2)); dividing 5 by 9 then subtracting this number from 1 results in 0.444 (on a scale of 0 to 1), meaning 44.4% similarity between the queries, to 1 decimal place.

$$S = 1 - \left[\frac{(\sum C_{3(i,j,k)}) / 2}{|C_3|} \right] \quad (2)$$

This process is now reproducible on-demand. Two queries can be consumed and a single similarity score can be produced.

Next, one must consider how these new functions to compare some incoming query A with all queries in the query cache can be used, ranking the latter using K-nearest neighbour to isolate the most similar queries. In doing so, the metadata can be checked on each of those queries and a majority vote conducted to determine which sub-schema selection is most appropriate for query A.

To do so, an independent query cache table is needed that can store metadata for use in the new process. In this query cache, the text, assigned schema ID, mapped query text, last execution duration and query weight for each query can be stored. These attributes will be used in the main body of the process.

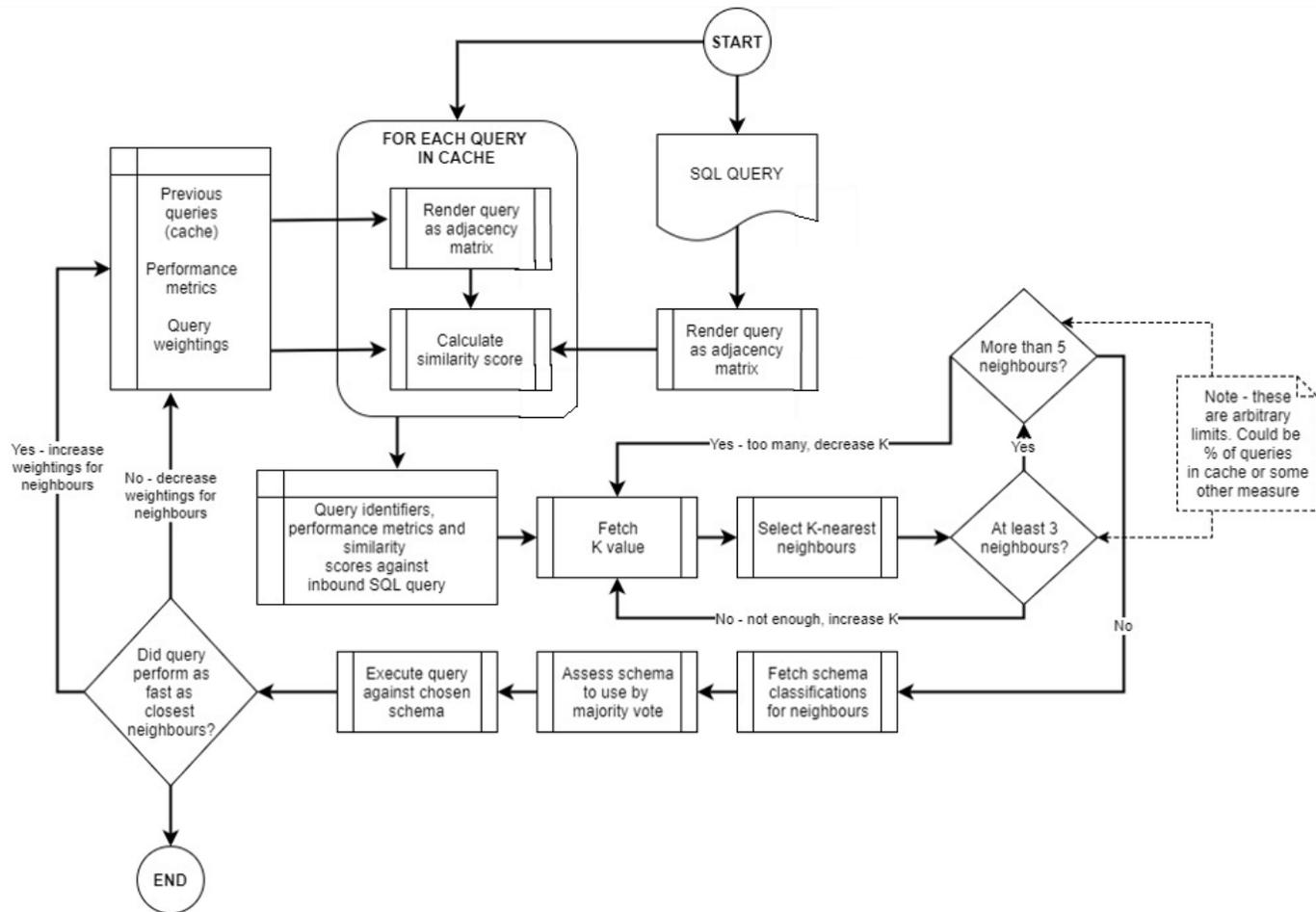


Fig. 7.6: Similarity scoring and query mapper process flow

Fig. 7.6 shows the process flow for the similarity scoring mechanism, classifier and query mapper.

The components not yet discussed, shown in Fig. 7.6, are the KNN weighted classifier and schema selector; these components extend the initial presentation of the solution in Chapter 5 and are detailed algorithmically in the next section before demonstration through code in section 7.3.

7.2 Algorithmic Implementation

It is now sought to integrate the similarity scoring process into the algorithms so far for PETAS. To do this, the algorithms presented in Chapter 6 are extended to allow for edge list and adjacency cube generation for two cubes. This is done by extending the inputs to consume sqlQueryA, sqlQueryB and a flag indicating which query is to be transformed into the edge list/adjacency cube. In doing so, code re-use is improved and wrapper code is simplified. However, the base algorithm remains the same; only the inputs change, so the algorithm is not re-presented here. Please see Chapter 6.

The similarity scoring mechanism requires algorithmic illustration. As inputs, it takes two complete adjacency cubes, passed as objects. Each object is a multidimensional list or array. As output, it computes a similarity score between 0 and 1 to 2 decimal places.

Algorithm 7.7 below shows the structure of this algorithm:

Algorithm 7.7: The similarity scoring algorithm

```
(inputs: cubeA of type object, cubeB of type object)
# calculate Hamming distance
initialise integer variable 'hamming' = 0
initialise integer variable cubeAEdgeCount = 0
initialise integer variable cubeBEdgeCount = 0
# begin dim-0 loop
for i in range 0 to the length of cubeA (max cubeA 0-dimension index):
--# begin dim-1 loop
----for j in range 0 to the length of the cubeA 1-dimension index:
-----# begin dim-2 loop
-----for k in range 0 to the length of the cubeA 2-dimension index
-----if cubeA (i, j, k) value is not equal to cubeB (i, j, k) value then
-----increment hamming += 1
-----if cubeA (i, j, k) value equals 1 then
-----increment cubeAEdgeCount += 1
-----if cubeB (i, j, k) value equals 1 then
-----increment cubeBEdgeCount += 1
-----# end dim-2 loop
----# end dim-1 loop
--# end dim-0 loop
initialise integer variable 'maxEdges', no value
set maxEdges to the max of cubeAEdgeCount, cubeBEdgeCount
initialise real number variable 'similarity' to no value
set similarity = hamming / 2.0 divided by maxEdges, rounded to 2 d.p.
return similarity to caller
```

This algorithm steps through the first cube of two on a dimension-by-dimension basis; for every value, the corresponding X-Y-Z co-ordinate in the second cube is looked-for and compared; if the values are not equal then the Hamming distance is incremented by 1. For efficiency, these loops also count the number of edges in the cubes and set the maximum of both as the denominator of the similarity scoring equation.

Next, the table structure of the new query cache used for classification and schema mapping is presented, shown in Table 7.8.

Table 7.8: Query cache table design

| Column Name | Data Type | Max Length/Value | Description |
|------------------------------|---------------|--|--|
| QueryID | INTEGER | 0-2 ³¹ -1 | Surrogate primary key; identity column. |
| QueryTextOriginal | NVARCHAR | 4,000 + MAX (row overflow) | Holds the original query text. |
| QueryWeight | DOUBLE/REAL | 0-2 ³¹ -1 (scale 16, precision 4) | Holds a real number representing query weight to be used in the KNN classifier. |
| AssignedSchemaID | SMALL INTEGER | 0-32768 | Pointer to the schema ID which ran this query most efficiently. |
| QueryTextNew | NVARCHAR | 4,000 + MAX (row overflow) | Holds the new, mapped query text to the indicated SchemaID. |
| LastExecutionDurationSeconds | INTEGER | 0-2 ³¹ -1 | Holds the last execution duration of the new query form in whole seconds, rounded. |

In a real implementation, the query cache would require populating with recently-executed queries from the inbuilt query cache (or re-execute the queries as they arrive, asynchronously, and collect the metadata). For the purposes of testing, this obliges the initial generation of a set of test queries, and the creation of a process to execute these test queries against a database, re-execute them against one or more alternative schemas, and collect the resultant metadata. This process is not described here as it is detailed in the next section; instead, a fully-populated cache table is assumed,, and the presence of both a database and a list of alternative sub-schemas available to select from is also assumed (Chapter 8 details this dynamic schema definition process).

Given the existence of the cache, the database and the schemas to select from, the KNN classifier is examined. This runs in real-time immediately after the adjacency cube generator; the KNN classifier calculates the similarity score for query A against all queries in the cache, identifies the closest-matching queries and selects the most appropriate schema.

Fig. 7.9 illustrates the concept of the KNN classifier. The beige circle is query A, the query at hand; all other circles are other, previously-run queries from the cache. Each pair of query A/query from the cache has a similarity score, calculated using this method. These scores are arranged on a 1-dimensional plane. An arbitrary K number of queries with the highest similarity scores to query A (here, $K=3$) ‘vote’ to assign a schema ID to query A; e.g., if query B has schema ID 1, query C schema ID 3 and query D schema ID 1, then the majority verdict is schema ID 1 and query A is executed against this schema.

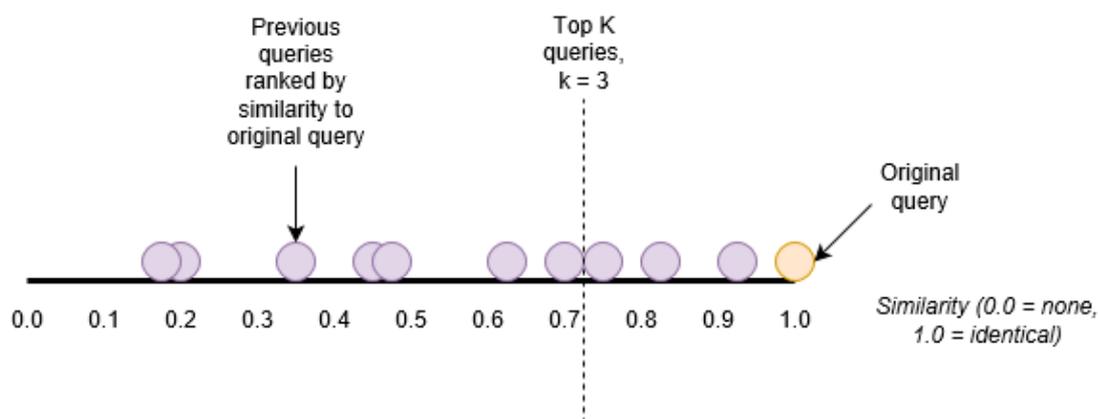


Fig. 7.9: The KNN classifier concept

Algorithm 7.10 shows first how the query cache is looped through, calculating similarity scores upon which to run the KNN classifier. The availability of the ‘similarity’ function (Algorithm 7.10) is assumed, and the cache table, named ‘querycache’, and a list is output with the query ID for each query in the table and the similarity score when compared to query A.

Algorithm 7.10: Looping through the query cache

```

input: queryA (SQL text of query in hand)

initialise new array/list 'comparison' with no elements
initialise new untyped variable 'similarity'
initialise new integer typed variable 'errorCount' and set to 0
initialise new integer typed variable 'queryXID' and set to nothing

for queryX in querycache:
  set currentQueryID to query ID of queryX in querycache table
  try:
    call function similarityFunction (queryA, queryX), output to 'similarity'
    write (queryXID, similarity) as [1,2] object to array 'comparison'
  catch:

```

```
increment errorCount += 1

return comparison
```

Algorithm 7.10 outputs an array/table consisting of two columns, queryXID and a similarity score (a real number).

This can then be used as input to the next stage, which is to find the most similar queries to the query at hand. At this juncture, weighting is introduced; every query in the query cache has a weight attached, defaulting to 1. The weight is looked up from the query cache table, before multiplying the similarity score by this weight and re-ordering the list. The weights are affected by how accurate or useful the query has been previously at correctly identifying a sub-schema where the query runs faster than against the base schema.

Algorithm 7.11 illustrates the process. The output is query execution for the caller and an entry into the query cache process with the last execution duration time.

Algorithm 7.11: Finding similar queries

```
input: 'comparison' array [2, n] comprising of a list of query IDs and similarity scores
to query A

for each query ID and similarity score pair in 'comparison':
--fetch the 'queryWeighting' for the query ID from the query cache
--multiply the similarity score by the query weighting
--write back the query ID and the resulting similarity score, overwriting the active pair
sort the array by descending similarity
define variable k as a typed variable and initialise to 3
define variable 'neighbours' as a typed array variable, empty
set neighbours to be the top K dim-0 elements in 'comparison' (query IDs), as ordered
define variable 'csv' as a typed list variable, empty
for each query id in neighbours:
--fetch the 'associatedSchemaID' matching query ID from the query cache
--append the schema ID as a new element in array csv
initialise variable 'verdict' as an empty untyped variable
set verdict to the element in 'csv' with the highest count (cardinality)
begin query timer
execute the query against the schema id specified in 'verdict'
end query timer
write back query execution metadata to query cache table
write back K nearest neighbours, query ID for query in hand and last execution duration
--to table 'querystack' for asynchronous assessment
```

Next, an asynchronous process is specified which reads the latest queries entered into the query stack table (the output of Algorithm 3). The query stack table is a temporary table which stores the query IDs of the nearest neighbours identified and the execution time of each query. The process uses this data to re-execute the query against the base schema and sends the results to /dev/null (no output). The query execution is timed. Should the query run faster against an

alternative schema than the base schema, the query weighting for the neighbours that specified the alternative schema is incremented by some constant, e.g. 0.1; else, in the vice versa case, the query weighting is decremented by 0.1. With repeated executions, this promotes query importance in the cache for queries that most often closely match inbound queries while naturally filtering out queries which are singular and do not routinely match inbound queries. Periodically, queries are removed from the cache that reach a certain negative threshold T.

Table 7.12 shows the table structure for the query stack and Algorithm 7.13 illustrates this process.

Table 7.12: Query stack table design

| Column Name | Data Type | Max Length/Value | Description |
|------------------------------|-----------|----------------------------|---|
| rid | INTEGER | 0-2 ³¹ -1 | Surrogate primary key; identity column. |
| queryTextOriginal | NVARCHAR | 4,000 + MAX (row overflow) | Holds the original query text. |
| queryTextNew | NVARCHAR | 4,000 + MAX (row overflow) | Holds the new query text for the query, mapped to the chosen schema |
| n1 | INTEGER | 0-2 ³¹ -1 | Pointer to the query ID of the first nearest neighbour |
| n2 | INTEGER | 0-2 ³¹ -1 | Pointer to the query ID of the second nearest neighbour |
| n3 | INTEGER | 0-2 ³¹ -1 | Pointer to the query ID of the third nearest neighbour |
| nk | INTEGER | 0-2 ³¹ -1 | Pointer to the query ID of the kth nearest neighbour |
| lastExecutionDurationSeconds | INTEGER | 0-2 ³¹ -1 | Duration of the last query execution in seconds, rounded |

Algorithm 7.13: The asynchronous query weight adjustment process

```
# runs periodically while table queue stack exists
# begin loop
if rows exist in table queuestack:
--fetch all queries from queue stack table into list 'queuestack'
--fetch current value of K as variable 'k'
--for each query in queue stack:
----fetch last execution time of query as 'lastExecutionTime'
----for n in range 1 to k:
-----fetch query ID, last execution time for query n from query cache table
-----if last execution time of n < lastExecutionTime:
-----set queryWeighting for query n in query cache table, decrement by 0.1
-----if last execution time of n > lastExecutionTime:
-----set queryWeighting for query n in query cache table, increment by 0.1
-----else do nothing
--pop query from queuestack
--goto loop start
```

A process could be introduced to adjust the constant K , which was initially set to 3. Looking for the top K queries that are most similar to the query (the nearest neighbours), a similarity value could be picked as the boundary condition, checking the neighbours inside the boundary and obtaining a majority verdict. However, it leaves no reason why K should change – K is irrelevant in this scenario, it is the query weightings that are the dynamic factor here, since each query weighting directly affects its similarity score's proximity to the test query. For this reason, K is set to start as fixed to some low odd-numbered constant such as 3, and the top K queries sorted by similarity score (descending order) are skimmed. The query weights are adjusted per execution, asynchronously. This fits in with the classical definition of KNN.

A routine is then defined that updates K like so – K gets bigger if the similarity scores returned by the queries tend to be high (i.e., 90th percentile). K is adjusted to reduce if the scores are low. This is on the basis that high similarity scores are most likely to return an accurate prediction of which schema to use, so the more of them taken into account, the more accurate and useful this process will be. Vice versa, if the scores are low, then if K is large then the potential for error in schema selection also increases. This is done asynchronously i.e., periodically regardless of how many queries are being processed.

Algorithm 7.14 illustrates the K -adjustment process. It is assumed K can be looked up from the data layer, for example as a constant in a control table.

Algorithm 7.14: Adjusting the value of K

```
define low threshold LT as a typed real number
define mid threshold MT as a typed real number
define high threshold HT as a typed real number
define lowK as a typed real number
define midK as a typed real number
define highK as a typed real number
# assuming existence of K in e.g. table 'kvalue'...
set LT = 0.6
set MT = 0.7
set HT = 0.8
set lowK = 3
set midK = 5
set highK = 7
fetch mean of similarity scores currently in cache
if mean >= LT and mean < MT:
--set K = lowK
if mean >= MT and mean < HT:
--set K = midK
if mean >= HT:
--set K = highK
```

Finally, a process is required to map the query in hand (queryA) to the schema ID chosen by the process. Several factors are relied upon here; first, that the similarity scoring algorithm will tend to choose schema recommendation queries similar to the query in hand and as such, the chosen sub-schemas will contain all the tables, columns and rows required to service the query. If this is not the case, then the base schema is chosen as default and the query executed as normal. Secondly, mapping is highly dependent on the schemas output by the dynamic schema mapping process (see Chapter 8). In the practical implementation, four sub-schemas are derived from a base schema by a simple 2-way sharding and partitioning algorithm to effectively quarter the data and it was found a large majority of queries were mapped correctly. This behaviour is expected to be exhibited in a real-world environment.

However, in Chapter 8 a more advanced query mapper component was presented where new schemas are generated according to query execution history, queries are mapped to the new schema versions and restructured to be syntactically valid, and where additional execution metadata is collected to create and destroy sub-schemas asynchronously such that there exist a constantly mutating set of sub-schemas from which the query selection mechanism can choose. For this reason, the presentation of the query mapper is deferred to the next chapter. In a full implementation, the appropriate mapped query can be chosen from the generated mapped query from the dynamic schema process or generated on-the-fly using the same methodology.

This concludes the algorithmic implementation of this process. Please refer to the flowchart in Fig. 6 for an overview of how all the components interact together. Section 7.3 presents the practical implementation of these components.

7.3 Practical Implementation

The code is first presented to calculate a similarity score from two input cubes. This is written in Python and corresponds to Algorithm 7.7. Sample wrapper code is also provided, demonstrating how to call each function in turn to move from SQL query, to edge list, to adjacency cube, and finally to similarity score given a second query.

The code listings are extensive and so are provided in Appendix D.

Next, an implementation of Algorithm 7.10 is presented, the process that loops through the query cache and calculates similarity scores for each pair of queryA and the member of the cache table at hand. This is a Python implementation using PostgreSQL as the data persistence layer, and is shown in Appendix D, Code Listing 2.

Next, similar queries are found for a given queryA and a set of similarity scores output by Appendix D, Code Listing 2, finding the most appropriate schema to run queryA against by majority verdict, execute the query and output the metadata to the query stack table and query cache table. This is done using Python and PostgreSQL for the data layer and the implementation is given in Appendix D, Code Listing 3, which maps to Algorithm 7.11.

Appendix D, Code Listing 4 shows the query cache, K-table and query stack table CREATE TABLE definitions in PostgreSQL.

Finally, the Python code for adjusting query weightings in the query cache table, reflecting Algorithm 7.13, is presented in Appendix D, Code Listing 5, written in Python for PostgreSQL.

In the next section, the test data set is described, together with the process of setting up the query tables, creating sample queries, creating sample sub-schemas, mapping the queries, setting weights, and configuring the environment. A working implementation for most of the design is presented, with some minor deviations and exceptions, and the experiments and outcomes are shown.

7.4 Experimental Design

PostgreSQL on Debian was chosen as the experimental framework, as the Debian platform offers side-by-side Python functionality (which is also installed) and the stack is entirely open-source which removes proprietary barriers and licensing concerns. The test environment is a Microsoft Azure virtual machine, size A0, with 0.75 cores allocation and 1GB RAM. This is a modest machine size chosen to highlight whether this process can be viable without excessive use of system resources.

The data set identified for testing is the same data set used for schema classification - the Chicago crime set, available for free in its raw form [1]. This data set was chosen as it has three principal advantages:

- It comprises of a sizeable amount of data which is more likely to take measurable time to execute against, increasing the accuracy of any test results
- It is a simple structure but can be split out to separate tables with relative ease
- It is interesting and current (updated daily)

The following link from the website allows for direct download of the data from the Public Safety dataset [1] via wget: <https://data.cityofchicago.org/api/views/ijzp-q8t2/rows.csv?accessType=DOWNLOAD>

This was downloaded and saved as /home/./chicago/chicagoRaw.csv.

The Chicago data is a single table split into 22 columns. There are (at the time of writing) ~6,490,000 rows of data. The file is ~1.42GB in size. The columns are described in Table 7.15:

Table 7.15: Description of the Chicago Public Safety data set

| Column Name | Description | Type |
|----------------------|---|-------------|
| ID | Unique identifier for the record. | Number |
| Case Number | The Chicago Police Department RD Number (Records Division Number), which is unique to the incident. | Plain Text |
| Date | Date when the incident occurred. this is sometimes a best estimate. | Date & Time |
| Block | The partially redacted address where the incident occurred, placing it on the same block as the actual address. | Plain Text |
| IUCR | The Illinois Uniform Crime Reporting code. This is directly linked to the Primary Type and Description. See the list of IUCR codes at https://data.cityofchicago.org/d/c7ck-438e . | Plain Text |
| Primary Type | The primary description of the IUCR code. | Plain Text |
| Description | The secondary description of the IUCR code, a subcategory of the primary description. | Plain Text |
| Location Description | Description of the location where the incident occurred. | Plain Text |
| Arrest | Indicates whether an arrest was made. | Checkbox |
| Domestic | Indicates whether the incident was domestic-related as defined by the Illinois Domestic Violence Act. | Checkbox |
| Beat | Indicates the beat where the incident occurred. A beat is the smallest police geographic area https://data.cityofchicago.org/d/aerh-rz74 —each beat has a dedicated police beat car. Three to five beats make up a police sector, and three sectors make up a police district. The Chicago Police Department has 22 police districts. See the beats at https://data.cityofchicago.org/d/aerh-rz74 . | Plain Text |
| District | Indicates the police district where the incident occurred. See the districts at https://data.cityofchicago.org/d/fthy-xz3r . | Plain Text |
| Ward | The ward https://data.cityofchicago.org/d/sp34-6z76 (City Council district) where the incident occurred. See the wards at https://data.cityofchicago.org/d/sp34-6z76 . | Number |
| Community Area | Indicates the community area where the incident occurred. Chicago has 77 community areas. See the community areas at https://data.cityofchicago.org/d/cauq-8yn6 . | Plain Text |
| FBI Code | Indicates the crime classification as outlined in the FBI's National Incident-Based Reporting System http://gis.chicagopolice.org/clearmap_crime_sums/crime_types.html (NIBRS). See the Chicago Police Department listing of these classifications at http://gis.chicagopolice.org/clearmap_crime_sums/crime_types.html . | Plain Text |

| | | |
|--------------|---|-------------|
| X Coordinate | The x coordinate of the location where the incident occurred in State Plane Illinois East NAD 1983 projection. This location is shifted from the actual location for partial redaction but falls on the same block. | Number |
| Y Coordinate | The y coordinate of the location where the incident occurred in State Plane Illinois East NAD 1983 projection. This location is shifted from the actual location for partial redaction but falls on the same block. | Number |
| Year | Year the incident occurred. | Number |
| Updated On | Date and time the record was last updated. | Date & Time |
| Latitude | The latitude of the location where the incident occurred. This location is shifted from the actual location for partial redaction but falls on the same block. | Number |
| Longitude | The longitude of the location where the incident occurred. This location is shifted from the actual location for partial redaction but falls on the same block. | Number |
| Location | The location where the incident occurred in a format that allows for creation of maps and other geographic operations on this data portal. This location is shifted from the actual location for partial redaction but falls on the same block. | Location |

(Table adapted from 'Columns in this dataset' [1])

First, a recipient table is created to stage the data from the CSV file. The columns have been slightly renamed to remove whitespace and avoid reserved words, and appropriate datatypes have been chosen where possible:

```
CREATE TABLE chicagobase (
  rid INTEGER,
  rcaseNumber VARCHAR,
  rDate TIMESTAMP,
  rBlock VARCHAR,
  rIUCR VARCHAR,
  rPrimaryType VARCHAR,
  rDescription VARCHAR,
  rLocationDescription VARCHAR,
  rArrest BOOLEAN,
  rDomestic BOOLEAN,
  rBeat VARCHAR,
  rDistrict VARCHAR,
  rWard INTEGER,
  rCommunityArea VARCHAR,
  rFBICode VARCHAR,
  rxCoordinate INTEGER,
  ryCoordinate INTEGER,
  rYear SMALLINT,
  rUpdatedOn TIMESTAMP,
  rLatitude DOUBLE PRECISION,
  rLongitude DOUBLE PRECISION,
  rLocation VARCHAR );
```

The data was loaded into the table using the \copy command in the psql client like so:

```
\copy chicagobase FROM '/home/del/chicago/chicagoRaw.csv'
WITH (FORMAT csv, DELIMITER ',', HEADER);
```

A base schema was created consisting of one table, chicagobase. Another schema is then created which splits the data horizontally (partitioning) and vertically (sharding) to create 4 tables as shown in Fig. 7.16, with the verticals linked on rID as primary key.

The partition tables will be called 'Alpha' and 'Beta' accordingly - Alpha before the midpoint of rDate, and Beta after. The shards will be called CrimeType and CrimeLocation. For example, the fourth table in Fig. 7.16 below is called 'CrimeLocationBeta':

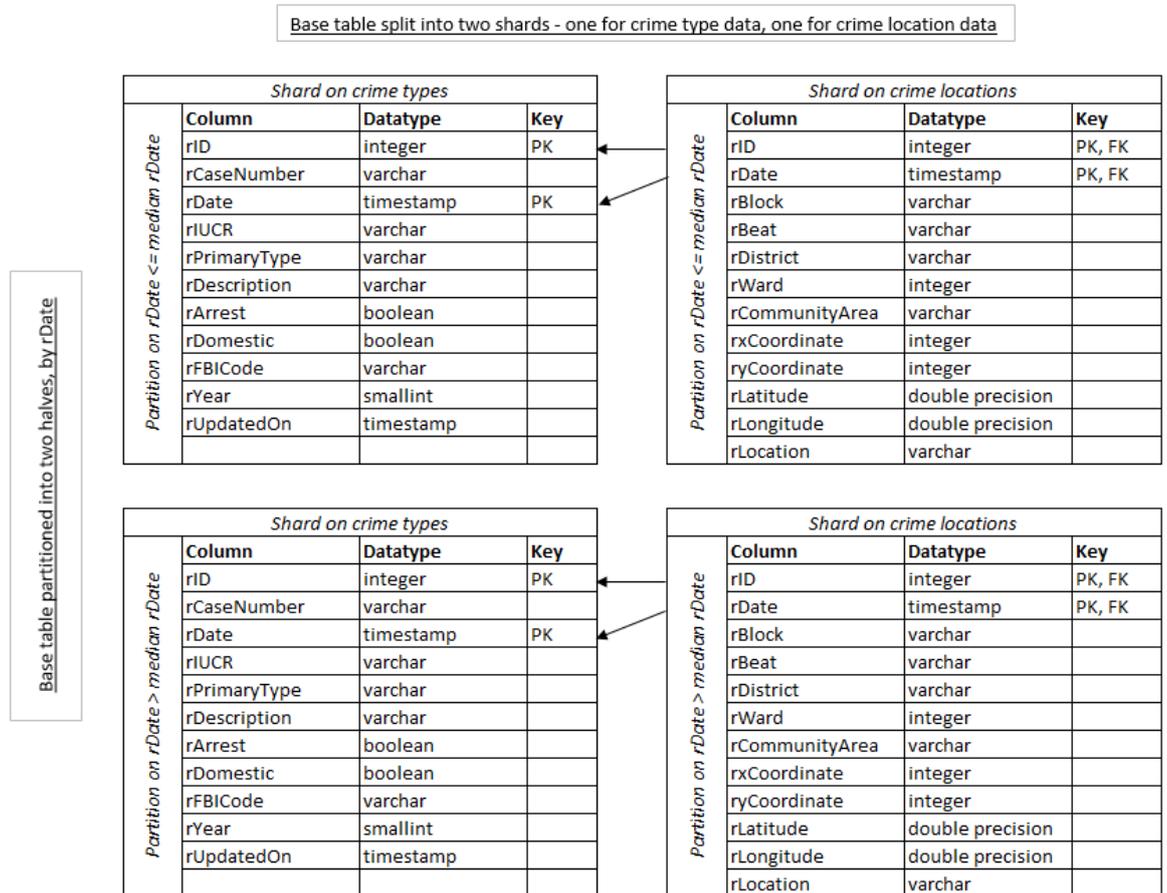


Fig. 7.16: Chicago data split into sub-schemas

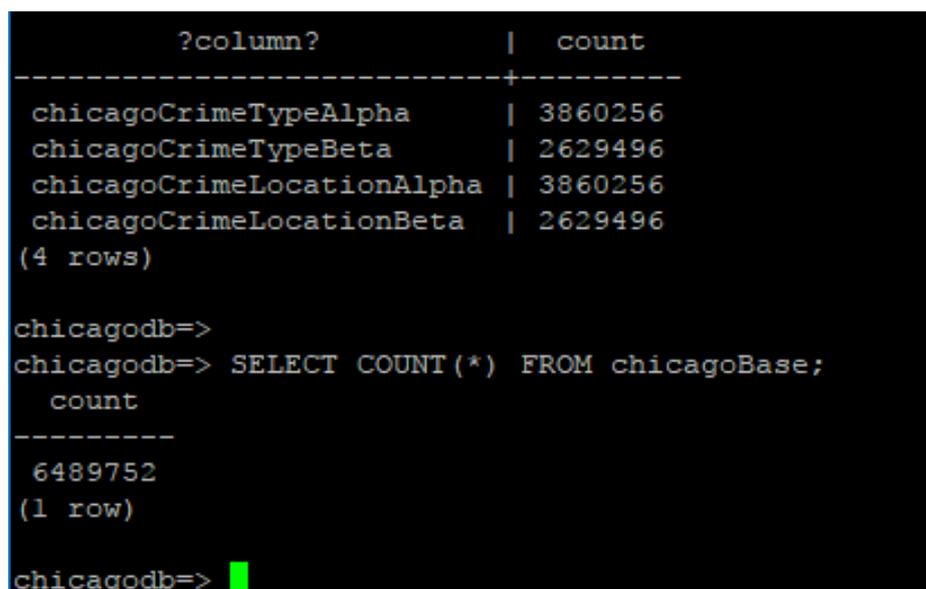
There are benefits to choosing this split:

- Queries which draw from only a selection of columns within a particular shard need use only the respective shard and not the full base schema
- Queries which draw only a limited set of row data may be able to use a particular partition rather than full scans of the base schema
- Crime type data (e.g. description, case number) is contextually separated from location data, which should result in better performance for queries which only need one or the other.

The statements to create and load these tables from the base schema in PostgreSQL are shown in Appendix D, Code Listing 6.

If a count is now issued of the populations of the tables, the data is shown to have been split between the two partitions alpha and beta (row count), and the two partitions, CrimeType and CrimeLocation (columns are split), making four tables in total. Fig. 7.17 shows the counts.

```
SELECT 'chicagoCrimeTypeAlpha', COUNT(*) FROM chicagoCrimeTypeAlpha
UNION ALL
SELECT 'chicagoCrimeTypeBeta', COUNT(*) FROM chicagoCrimeTypeBeta
UNION ALL
SELECT 'chicagoCrimeLocationAlpha', COUNT(*) FROM chicagoCrimeLocationAlpha
UNION ALL
SELECT 'chicagoCrimeLocationBeta', COUNT(*) FROM chicagoCrimeLocationBeta;
SELECT COUNT(*) FROM chicagoBase;
```



```
      ?column?          | count
-----+-----
chicagoCrimeTypeAlpha | 3860256
chicagoCrimeTypeBeta  | 2629496
chicagoCrimeLocationAlpha | 3860256
chicagoCrimeLocationBeta | 2629496
(4 rows)

chicagodb=>
chicagodb=> SELECT COUNT(*) FROM chicagoBase;
 count
-----
 6489752
(1 row)

chicagodb=>
```

Fig. 7.17: Table cardinalities in the Chicago sub-schemas

K-nearest neighbour is an interval-based machine learning classifier. These types of classifiers can be used in unsupervised learning; however, the approach used here is a slight modification - it is a selection of neighbours using KNN, but a majority verdict of the classification decision to make based on the classification decisions of those selected neighbours. Therefore, a set of labelled training data already in the cache is required - the more queries, the better; also, the weights need to be pre-set, and likewise the value of K.

For any query, there are two schemas to choose from - the single-table schema and the four-table schema. For testing purposes, some queries are required (to be written or generated) on one of these schemas. These are written on the single-table schema first before deriving the 4-table equivalents, since this will allow an opportunity to fully test the similarity algorithm in both directions.

These queries are then labelled by hand with what are believed to be the most appropriate schema for it; then this information is recorded into the cache. This produces a set of training data.

It was necessary to write a random query generator specifically for this data set. This query generator was mentioned in an earlier chapter, and outputs from it were used to test the query representation algorithms. This is presented in full in Appendix D, Code Listing 7. SQL Server was used, as PostgreSQL did not have query variable support, and it was necessary to construct SQL dynamically and with complex methods such as side-effecting random variables.

This query generator is used for test purposes and does not form part of the novel contribution to knowledge, so the algorithm is not presented here; in brief, it generates random values to fit a variety of domains and data types for a series of columns passed into it from the Chicago base tables.

Next, for the purposes of testing, a new stored procedure was created which would generate the equivalent query against the alternative schema (the version with 4 tables). To do this, a determination of whether columns in the query belonged to crimeType, crimeLocation or both was made; and it was determined, in the case where rDate was a predicate, whether the date indicated fell before, or after, the median (so as to determine whether to use the alpha partition or beta partition). If rDate isn't a predicate then a UNION ALL is necessary.

Again, using SQL Server, this mapping was possible programmatically as shown in Appendix D, Code Listing 8. A table-valued function (a function that returns a result set, also known as a TVF) was created that takes a single statement as input, creates the alternative schema, and returns both the original and new statements as output.

This function is used in a CROSS APPLY to generate as many queries as necessary; the example in Appendix D, Code Listing 9, written in T-SQL, uses the function above to generate 1,000 queries. Fig. 7.18 illustrates the output.

| rid | stmt | alt |
|-----|---|---|
| 1 | SELECT rCommunityArea, rcaseNumber FROM chicagoBase ; | SELECT rCommunityArea, rcaseNumber FROM chicagoCrimeTypeBeta a INNER JOIN r |
| 2 | SELECT rWard, rcaseNumber, rLatitude, rUpdatedOn, rDistrict, rCommunityArea, rDate, rY | SELECT rWard, rcaseNumber, rLatitude, rUpdatedOn, rDistrict, rCommunityArea, rDa |
| 3 | SELECT ryCoordinate, rArrest, rLocation, rCommunityArea, rYear, rDate FROM chicag | SELECT ryCoordinate, rArrest, rLocation, rCommunityArea, rYear, rDate FROM chicag |
| 4 | SELECT rDescription, rBeat, rLongitude FROM chicagoBase ; | SELECT rDescription, rBeat, rLongitude FROM chicagoCrimeTypeBeta a INNER JOIN cl |
| 5 | SELECT rcaseNumber, rIUCR, rDistrict, rBeat, rDescription, rid FROM chicagoBase ; | SELECT rcaseNumber, rIUCR, rDistrict, rBeat, rDescription, rid FROM chicagoCrimeTyp |
| 6 | SELECT rIUCR, rBeat, rPrimaryType, rDate FROM chicagoBase ; | SELECT rIUCR, rBeat, rPrimaryType, rDate FROM chicagoCrimeTypeBeta a INNER JOIN |
| 7 | SELECT rDate, rDomestic, rDescription, rDistrict, rLocation, rFBIcode FROM chicagoBase W NULL | |
| 8 | SELECT rid, rBeat, rCommunityArea, rPrimaryType, rFBIcode FROM chicagoBase ; | SELECT rid, rBeat, rCommunityArea, rPrimaryType, rFBIcode FROM chicagoCrimeTyp |
| 9 | SELECT rDate FROM chicagoBase WHERE rDate = '2014-10-06 12:00:24' ; | NULL |
| 10 | SELECT rWard, ryCoordinate, rIUCR, rBlock, rxCoordinate, rDistrict, rBeat FROM chicag | SELECT rWard, ryCoordinate, rIUCR, rBlock, rxCoordinate, rDistrict, rBeat FROM chica |
| 11 | SELECT rLocation, rDate, rid, rBeat, rLocationDescription FROM chicagoBase ; | SELECT rLocation, rDate, rid, rBeat, rLocationDescription FROM chicagoCrimeLocatio |
| 12 | SELECT rFBIcode, rArrest, ryCoordinate, rPrimaryType, rid, rBeat, rxCoordinate, rDescripti | SELECT rFBIcode, rArrest, ryCoordinate, rPrimaryType, rid, rBeat, rxCoordinate, rDes |
| 13 | SELECT rWard, rYear, rcaseNumber, rxCoordinate, rid, rPrimaryType, rLongitude, ryCoor | SELECT rWard, rYear, rcaseNumber, rxCoordinate, rid, rPrimaryType, rLongitude, ryCo |
| 14 | SELECT rDomestic, rBlock FROM chicagoBase ; | SELECT rDomestic, rBlock FROM chicagoCrimeTypeBeta a INNER JOIN chicagoCrimeLu |
| 15 | SELECT rPrimaryType, rxCoordinate, rid, rDescription, rLocation, rIUCR, rLongitude, rBeat | SELECT rPrimaryType, rxCoordinate, rid, rDescription, rLocation, rIUCR, rLongitude, r |
| 16 | SELECT rBeat, rLocationDescription, rIUCR, rLatitude, rcaseNumber FROM chicagoBase ; | SELECT rBeat, rLocationDescription, rIUCR, rLatitude, rcaseNumber FROM chicagoCri |
| 17 | SELECT rPrimaryType, rcaseNumber, rxCoordinate, rWard, rYear, rDescription, rLocation | SELECT rPrimaryType, rcaseNumber, rxCoordinate, rWard, rYear, rDescription, rLocat |
| 18 | SELECT rid, ryCoordinate, rcaseNumber, rWard, rIUCR, rYear, rLatitude, rLongitude, rPr | SELECT rid, ryCoordinate, rcaseNumber, rWard, rIUCR, rYear, rLatitude, rLongitude, r |

Fig. 7.18: Output from the random SQL query generator

From here, the training data requires importing into the QueryCache table along with some other information – query weightings, which are all initially set the same; and NULL for query execution time. A decision is needed for each query on which schema would be most appropriate for the query, which was done automatically according to the following heuristics.

Rule 1

IF query uses BOTH partitions (alpha and beta)
AND query uses BOTH shards (type and location):

Use base schema

(on the basis that no savings will be made using the 4-table schema so the base schema will be quicker)

Rule 2

IF query uses BOTH partitions (alpha and beta)
AND (query uses type shard XOR query uses location shard):

Use base schema

(on the basis that the UNION ALL is redundant and so the base schema will be quicker)

Rule 3

IF query uses the alpha partition XOR query uses the beta partition
AND query uses BOTH shards (type and location):

Use 4-table schema

(on the basis that the row count is divided in 2 so the seek time should be lower across the rows)

Rule 4

IF query uses the alpha partition XOR query uses the beta partition
AND query uses the type shard XOR query uses the location shard:

Use 4-table schema

(on the basis that the 4-table schema presents the smallest possible set so should be quicker)

First, the data was imported, leaving aside `lastQueryExecutionTime` and `AssignedSchemaID`, setting all weights to 1, to the `QueryCache` table in PostgreSQL:

```
INSERT INTO querycache
  SELECT t.rid, t.stmt, 1.0, NULL, t.alt, NULL
  FROM   public.trainingdataraw AS t
  ORDER BY t.rid;
```

Finally, a cursor was created which would loop through all the query pairs now in `QueryCache`. For each query pair, the cursor would a) select a schema using the rules above and b) execute the requisite query (for the schema) and finally record the query execution time in `LastQueryExecutionTime`. In this way, the `QueryCache` table was populated and the training data is ready to use.

The test data is stored in the `TestDataRaw` table for use during the testing, documented in the next section.

7.5 Testing and Results

Several subprocesses have been defined that form the similarity scorer and schema mapper. In this section, the tests are specified and the results are shown for various units within the process and for the process in the main.

Firstly, the similarity scoring mechanism is tested with 5 pairs of queries, to get an indication on whether this process is viable. The queries are listed in Appendix D, Code Listing 10, in a test harness written in Python against the similarity function.

These queries were chosen (also in column 2, Table 7.20) to illustrate a range of similarities. The first query pair are structurally and functionally identical; the final query pair are structurally similar but the objects are completely dissimilar and so should not generate a high similarity score.

The results are shown in Table 7.19.

A low deviation in the results from the expected scores (the hypotheses) is noted, indicating optimism that the algorithm is returning results in an expected range.

Table 7.19: Results from similarity scoring process testing

| sqlQueryA | sqlQueryB | Expected score* | Actual score | Delta |
|--|---|-----------------|--------------|-------|
| SELECT A.x, B.x FROM A INNER JOIN B ON A.x = B.x; | SELECT A.x, B.x FROM A INNER JOIN B ON A.x = B.x; | 1 | 1 | 0 |
| SELECT A.x, B.y FROM B INNER JOIN A ON A.z = B.z; | SELECT A.x, B.z FROM B INNER JOIN A ON A.z = B.z; | 0.8 | 0.75 | 0.05 |
| SELECT A.x, A.y, A.z FROM A INNER JOIN B ON A.y = B.y; | SELECT B.x FROM A INNER JOIN B ON A.y = B.y WHERE A.x = 10; | 0.5 | 0.67 | 0.17 |
| SELECT A.x FROM A INNER JOIN B ON A.x = B.x WHERE B.y > 100; | SELECT B.y FROM A INNER JOIN B ON A.z = B.z WHERE A.z = 0; | 0.2 | 0.29 | 0.09 |
| SELECT A.x FROM A INNER JOIN B ON A.x = B.x WHERE A.x = 10; | SELECT C.x FROM C INNER JOIN D ON C.z = D.z WHERE D.z > 50; | 0 | 0 | 0 |
| | | | Mean | 0.062 |
| | | | St. dev. | 0.071 |
| | | | Range | 0.17 |

* Note: Expected score is an estimate

Some limitations were noted with the implementation, particularly:

- No support for nested queries e.g. subqueries or CTEs
- Limitation on complex JOIN and WHERE conditions
- WHERE clauses limited to AND or OR (no support for constructs like BETWEEN or IN)

The suitability of the solution for the full range of allowable ANSI-SQL is discussed in the conclusions.

Next, the query generator function was tested against 10 sets of 1,000 queries (which were generated using the random query generator function), with the aim to discover how many, if any, alternative mapped queries failed to be generated by this process; or, the failure rate. Table 7.20 shows the results.

Table 7.20: Failed query mappings

| Run # | NULL count |
|---------------|------------|
| 1 | 41 |
| 2 | 53 |
| 3 | 41 |
| 4 | 50 |
| 5 | 50 |
| 6 | 50 |
| 7 | 56 |
| 8 | 34 |
| 9 | 45 |
| 10 | 48 |
| <i>Mean</i> | 46.8 |
| <i>Median</i> | 49 |

With each run consisting of 1,000 queries, the average failures were 46.8 queries per 1,000 queries; a failure rate of 4.7%. This is an optimistic result, as the converse view is that 95.3% of queries were mapped successfully. More work is required on the implementation to converge this percentage to 100%.

Next, it was observed if the queries generated and their alternatives all executed correctly, i.e. they are syntactically and functionally valid. To do this 4-table schema was recreated in Microsoft SQL Server (empty of data), then a cursor was used to iterate over each query pair, executing each in turn. If both executed without erroring, the query pair was marked as valid.

The SQL code to do this is shown in Appendix D, Code Listing 11.

It was here that some severe issues with the method were noticed. The ‘good’ queries numbered only 163 of 1000 (16.3%), a failure rate of 83.7%. There were periodic system crashes as the system struggled to cope with executing 1,000 queries and rendering the result sets. It was determined that the problem was that some columns SELECTed in the alternative query didn’t exist in the shards of the table selected (i.e. ‘rYear’ exists in the type shard, not the location shard so a query against the location shard that specifies this column would fail).

After some consideration, it was realised there were several problems – the first problem lay in the shard flag settings of my query generator – rYear was missing. The following line was added:

```
OR @inboundQuery LIKE ('%rYear%')
```

The second problem was the LocationDescription column was missing from the LocationAlpha/Beta tables. This was added in a similar manner.

The third problem was that code after the UNION ALL was JOINing between the LocationBeta and LocationBeta tables instead of the TypeBeta and LocationBeta tables, a result of a simple typographical error. This section was amended to:

```
REPLACE(@outboundQuery, 'chicagoCrimeTypeBeta a
    INNER JOIN chicagoCrimeLocationBeta b ON a.rid = b.rid',
    'chicagoCrimeTypeBeta a
    INNER JOIN chicagoCrimeLocationBeta b ON a.rid = b.rid')
```

The fourth problem was that the 'rid' column was ambiguous when included in the SELECT, since aliases are not being used. The code was amended so all FROMs were aliased, and a section was added to explicitly replace rid and rDate SELECTs with the same plus appropriate aliases. This is not an elegant solution but given this code doesn't form a core part of the solution (only the test harness) the workaround does not undermine the design.

The fifth problem was the second half of queries containing UNION ALL was identical in some circumstances to the first half of the query. This was traced back to two mis-specified @variables, and fixed this issue.

The sixth problem was that some queries were generated that broke data typing rules i.e. with WHERE predicates like this - ... WHERE rid = 'some string' when rid is an INT. This was due to the omission of a line dealing with the INT datatype in the QueryGenerator procedure, which was subsequently added.

The seventh problem was the occasional appearance of the single quote ' in literals used in the WHERE clause. Adding a REPLACE clause to replace in-data instances of single quotes fixed this issue.

The test query count to was lowered 100 to counteract the system resources problem, batching it to run 10x times to get the 1,000 queries desired per run.

Retesting with these fixes, this yielded a failure rate of nil; or 100% 'good' queries, discounting NULL-valued alternatives.

These were exported to PostgreSQL in the test instance. 954 queries were exported by way of a training set, and another 955 for testing purposes, overcoming the crashing issue when generating queries by executing at the command line. Export to flat file was done via the Import and Export utility in SSMS (SQL Server) from the dbo.TrainingData and dbo.TestData tables (where the output was stored from the above). The data was then imported using the tools provided by DataGrip (the PostgreSQL IDE in use) directly from the flat files.

Next, with the training and test data sets in PostgreSQL, the test outcomes from running the process end-to-end are presented; both the synchronous element (real-time query processing using adjacency cube generator, similarity scoring, schema selection and query mapping), and the asynchronous element (updating query metadata including weightings and execution times).

These tests were conducted using the scientific method. Table 7.21 describes the battery of end-to-end tests.

Tests 1, 2 and 3: 10 new SQL queries were generated and the matrix parser implementation was ran against them using the whole metadata cache, running the process at the individual level, whole-query level and whole-batch level to ascertain timings, which amounted to 9,520 executions of the algorithm. The processing time was found to be highly variable, with a mean average of 54ms per Q/Q_x comparison and a standard deviation from the mean of 25ms. Error handling was introduced in the test harness but errors at this stage were nil. The range of the durations varied between 278ns and 158ms. These results are shown individually and grouped by query in the diagrams in Figs. 7.22(a) and 7.22(b). The variance between queries is clearly visible by the column height differences shown in Fig. 7.22(a) and the differences in the mean markers in Fig. 7.22(b).

Test 4: This was to ascertain whether queries running under PETAS executed faster on average than queries using only the normal execution process. This test was scoped – the aim was to measure whether using schema selection resulted in overall faster execution, rather than testing the end-to-end process. Tests 1-3 highlighted an issue in the implementation of the classifier – queries were taking, on average, 54ms to be compared against each neighbour, the delay mostly due to iteration when parsing the query into the matrix. This scaled up to a significant and unviable delay per query execution. It is envisioned that further development of this functionality could result in significant improvements, for example by storing matrices in the metadata rather than enforcing recalculation; limitation to some n sample of potential neighbours rather than the full set; rework of the algorithm implementation to use parallel threads; and looking into more efficient mathematical models for matrix calculations.

Table 7.21: Test descriptions

| Test # | Component tested | Description | Test Method | Measurement | Positive hypothesis | Configuration & Parameters |
|--------|--------------------------------------|--|--|-------------|---|--|
| 1 | Matrix parser | Speed test to measure how fast an individual query can be compared to another query using the matrix parser. | Call the matrix parser against query Q and Qx. | Time | Process completes without error. | 10 test queries (Q) used against a metadata cache of 952 queries (Qx). |
| 2 | Matrix parser | Speed test to measure how fast an individual query can be compared to a range of other queries using the matrix parser. | Call the matrix parser against query Q and a range of Qx. | Time | Process completes without error. | 10 test queries (Q) used against a metadata cache of 952 queries (Qx). |
| 3 | Matrix parser | Speed test to measure how fast a set of queries can be compared to a range of other queries using the matrix parser. | Call the matrix parser against a range of Q and a range of Qx for each Q. | Time | Process completes without error. | 10 test queries (Q) used against a metadata cache of 952 queries (Qx). |
| 4 | KNN Selector and Schema Classifier | Test to see if PETAS results in a set of queries running faster against a multi schema environment than against a single schema environment. | Take query execution timings for test queries against the base schema, then re run test queries using PETAS allowing schema choice, and compare the timings. Overhead for PETAS to be disregarded for this test. | Time | On average, queries run faster when given a choice of schemas (under PETAS) than when given a single base schema. | 952 test queries (Q) used against a metadata cache of 50 randomly-drawn queries from a population of 952 (Qx). |
| 5 | Query Mapper | Execution of queries using PETAS is successful. | A set of test queries used against a metadata cache of queries, with asynchronous feedback loops running regularly. | Error count | All queries run successfully. | 952 queries (Q) tested. |
| 6 | KNN Selector and Feedback Mechanisms | Query weights (W) in the metadata cache will fluctuate in proportion to query executions using PETAS. | Fetch the query weights from the metadata cache after all test queries are executed. | The set W | Some values of W fall in the range $W < 1$ and $W > 1$. | 86 new test queries (Q) used against a metadata cache of 952 queries (Qx). |
| 7 | KNN Selector and Feedback Mechanisms | Similarity scores will improve in proportion to the number of query iterations. | For each test query in a set in order of execution, measure the average similarity score for their neighbours. | Time | Mean similarity scores increase in a positive correlation with the number of queries (Q) processed. | 86 new test queries (Q) used against a metadata cache of 952 queries (Qx). |

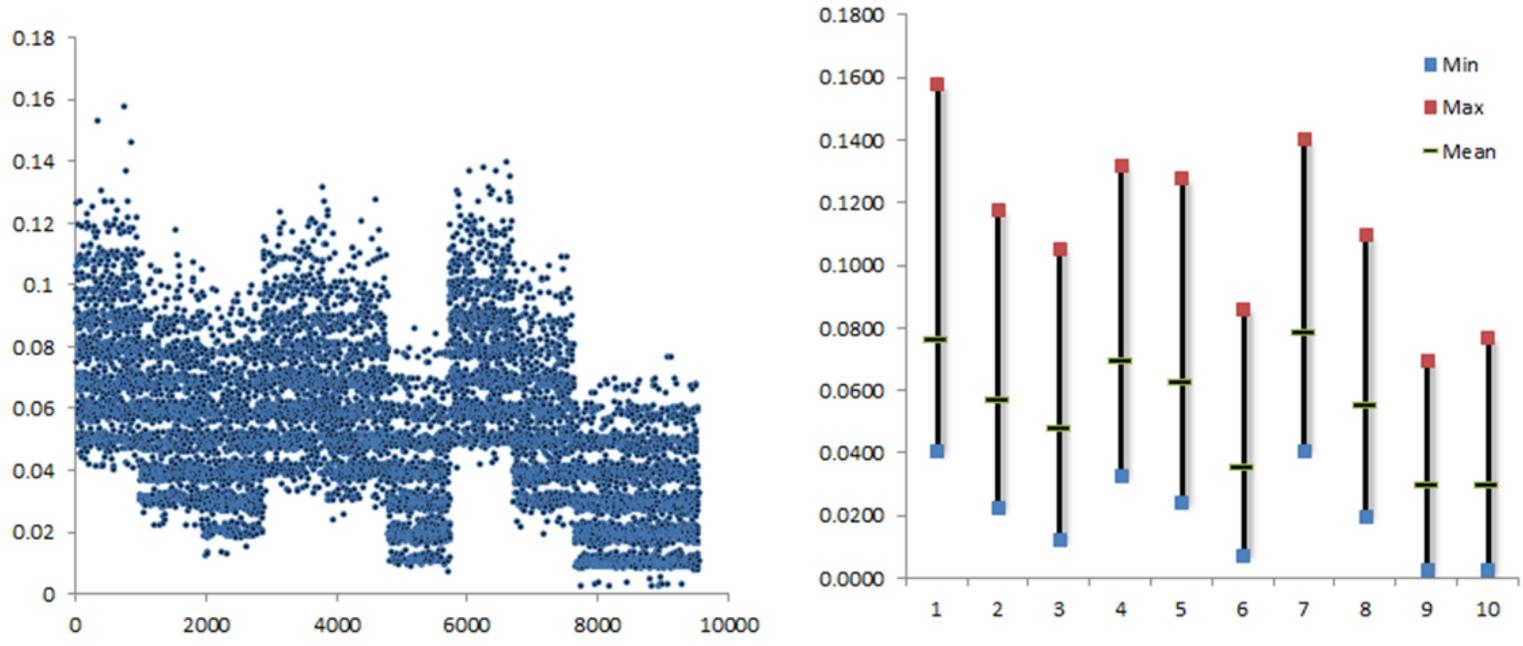


Fig. 7.22(a) and 7.22(b): Processing time (ms) per query and per run (batch)

However due to this 54ms delay, it was necessary to exclude from scope this overhead for test 4. The number of Q/Q_x comparisons were also limited to 50, selected randomly from the cache. Doing so meant that a) it was less likely that an improvement in query classifier accuracy would be recorded because the K -nearest neighbours in a sample of 50 would be less accurate (have lower scores) for Q/Q_x than the K -nearest neighbours in the full query set; and b) any multiple increase in individual query weights would be dependent on the query being sampled more than once. By sampling 50 from approximately 950 queries, the probability of selection is approximately 0.052 rather than 1, therefore dampening any classifier improvement as a result.

Using the pool of 952 test queries Q against 50 Q_x queries drawn randomly (47,600 comparisons), it was found that 47.9% of queries were classified to the alternative schema and the remainder to the base schema. In order to ensure test validity, the timings for all queries were re-ran against the base schema only and an unexplained deviance was noticed in the average query execution time of +9.9%, consequently labelled D and corrected for in the analyses. The deviance is attributed to unrelated background operating system activity stemming from the use of cloud, rather than fixed, computing resources.

The results of running these queries were a mean reduction in query execution time of 6.2% for all queries regardless of schema assignation, and a reduction in query execution time of 20.6% for queries executed against the alternative schema. Fig. 7.23 shows these cost savings for all queries. The upper trendline indicates the mean original query execution time (with D correction) and the lower trendline indicates the mean query execution time with schema selection.

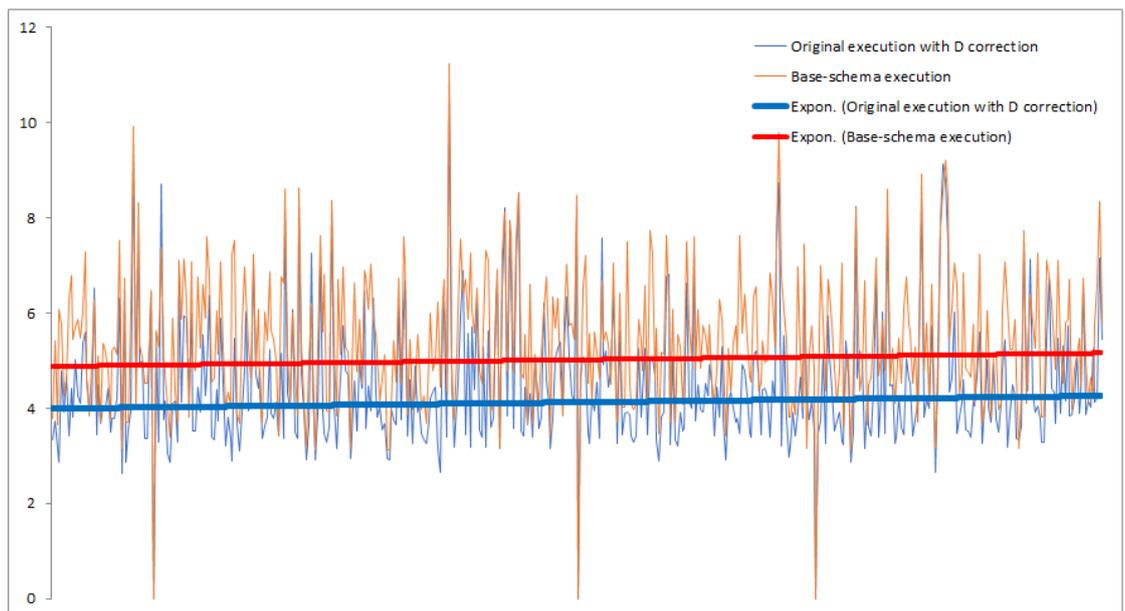


Fig. 7.23: Cost savings (execution time in ms) per query, per schema

Test 5: This test was to check the error count of the test queries from test 4. Ideally there would be no errors in execution. There were 8 queries of 952 found to be errored due to syntactic issues, a rate of 0.84%. This reflects a limitation in the proof of concept, since as previously discussed, a full PETAS implementation would not rely on syntactic mapping but would map at the parse tree or other lower level.

Test 6: This test aimed to examine whether query weights in the metadata cache were being adjusted as part of the end-to-end PETAS process. Some implementation issues were found which caused queries to be deleted from the cache before their weights were adjusted and these errors were corrected. All weights were set to 1. KNN was calculated by multiplication of S by W for each Q/Q_x tuple. Where a weight was adjusted, it was incremented or decremented by 0.1 for all current Q_x in K . The metadata cache was kept at its previous population of 952 and generated 100 new test queries. 86 were syntactically valid, where 14 failed validation (due to weak implementation of the query mapper). The 86 were ran through the PETAS process. It was found that in 30 cases, members of the cache (Q_x) were being incremented or decremented, of which 4 cache members' weights were adjusted more than once, and the range of W_x varied between 0.8 and 1.2. Increments and decrements were evenly split.

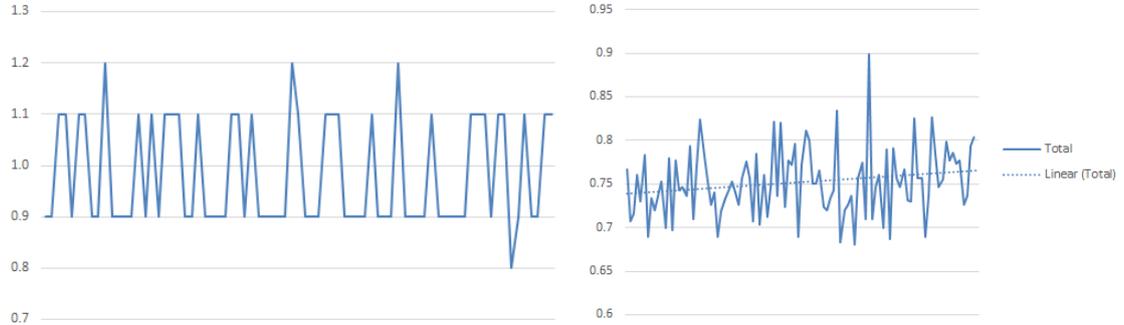
This result is important because the feedback mechanism of the classifier relies on the query weights being adjusted, either positively or negatively. Without this feedback, PETAS would be static and would not continually learn from new input. With 86 test queries (Q) and $K = 3$, it was expected that a maximum of 252 queries in the cache (the pool of Q_x) would have their weights adjusted. That this adjustment happened, and that a proportion of the queries in the metadata cache (30 different Q_x were affected) means that the KNN mechanism is working – queries (Q_x) are being selected from the cache corresponding to the structural similarity to inbound queries (Q) and furthermore, are being selected multiple times, as evidenced by the ratio of 30 to the maximum 252. Test 7 discusses how the similarity scores are affected when the same queries are being selected from the cache.

Test 7: This final test aimed to establish whether the KNN classifier was improving its own accuracy through weight adjustment. Such an improvement would manifest in mean average similarity scores from the matrix parser for successive Q/Q_x combinations increasing over query iterations, as the 'useful' queries' chances of selection were probabilistically increased by weighting.

The same 86 queries Q were used as selected in Test 6, and K was set to 3, obtaining 3 similarity scores S_1 , S_2 and S_3 for pairs Q/Q_1 , Q/Q_2 and Q/Q_3 for each Q (a total of 258 similarity scores). All S_x outcomes ranged between 0.68 and 0.9.

A modest positive correlation was found between successive query iterations and S . Using the slope-intercept method, the function of this correlation can be calculated as $y = 0.00035294x + 0.74$

(see the trend line in Fig. 7.24(b)). As an aside, using this linear formula as an approximation, it is possible to predict the number of queries required to be processed to achieve a specific S average (for y values < 1): e.g. for $S = 0.98$, this formula yields an estimate of 680 queries. Further testing would be required to establish the limits of this process.



Figs. 7.24(a) and 7.24(b): Query weight distribution, and query iteration correlation to similarity, respectively.

This concludes the testing and results. In the next section, the conclusions from the tests are briefly summarised and their impact on the viability of the solution design is discussed. Conclusions are discussed more broadly in the final chapter.

7.6 Conclusions

The testing carried out demonstrates the functionality of the matrix parser and KNN classifier, which worked as designed and demonstrated that query performance can be improved by matching queries to the most appropriate schema in an approach using multiple logical data representations. Limitations were observed; the implementation and testing did not support the full ANSI standard, and the existence of an unacceptable overhead during execution was evident. It is believed that these issues can be overcome by implementation improvements, for example by pre-calculating adjacency cubes and storing these in the metadata cache, and by replacing loop-based syntactic parsing methods. PETAS was demonstrated to work from the query parser through to the adjacency cube generation (Chapter 6), then through the similarity scoring mechanism, schema selector and query mapper (Chapter 7), resulting in a significant improvement of query execution times exceeding 20% (for over 50% of a test population) through the presentation of a choice of schemas and the activities of the new machine learning-led classifier. It was also demonstrated that PETAS learns from experience, with the constant adjustments of weights leading to more accurate

query classifications and a general increase in similarity scoring, although correlations of the latter were weak.

7.7 Chapter Summary

In this chapter, the work on the transformation process from SQL query to adjacency cube in Chapter 6 was extended, and it was shown how the new ML-driven functions can be used to calculate relative similarity between two queries, or cubes, how to use this similarity measure in a KNN implementation to find the most similar queries from a cache to a given query, and how to select the most appropriate schema by majority verdict. The process to dynamically adjust K was demonstrated and how query weighting can be used to give precedence to those queries which yield schema selections that most often result in decreased execution times, resulting in a self-learning methodology. The implementation of the same components was presented, built using Python on Debian, PostgreSQL, and Microsoft SQL Server. Limitations on the implementable features were noted, including lack of support for the full SQL standard, and the investigation showed how the results reflected potential performance improvements but only to a limited subset of all queries, and with a performance overhead manifesting as increased execution time that requires tuning out via an improved implementation.

Chapter 8 presents a dynamic schema definition process which monitors inbound queries to the database engine, uses the metadata in the query cache to create sub-schemas based on demand, de-allocates and destroys underused sub-schemas, and presents an alternative query mapper implementation. This component can be used alongside the work presented in this chapter to provide various alternative sub-schemas in an asynchronous fashion, meaning manual setup of sub-schemas as demonstrated has the potential to be fully automated. In this way, all components of PETAS become fully automatable, improving the viability of this solution for further development into industry-standard RDBMS tooling.

Chapter 8 – Testing: Dynamic Schema Redefinition

Please note that this chapter is a revised version of the research published in Colley and Asaduzzaman, 2020 [1]. Omitted code listings are provided in Appendix E.

8.1 Introduction

As the Zermelo-Fraenkel set-theoretic axioms as applied to Codd’s relational model allow for the expression of subsets from base sets using the axiomatic schema of separation [2], the axiom schema of separation can be extended into the relational database space by specifying and prototyping a new cross-platform technique using materialised views (MVs) for rapid, real-time schema derivation to reduce the query space and improve the query cost and resource use of database queries for a faster, more efficient transactional throughput.

From the research and solution design, it is concluded that MVs may present a potentially viable solution to describing, persisting and using subset data sets as alternative derivations from the base schemata and used in conjunction with the query cache as an asynchronous process, provide the opportunity for dynamic, real-time schema derivation for better query performance.

8.2 Algorithmic Implementation

A high-level overview of the key components of the algorithms comprising the dynamic schema redefinition element of PETAS and their interfaces with the existing RDBMS query processor are illustrated in Fig. 8.1 (new components are in the dashed area).

Three new processes are introduced to implement dynamic schema redefinition, dependent upon two new global temporary tables. The query parser fetches queries from the plan cache and divides their syntax into attributes, data sources and predicates (SELECT, FROM and WHERE subclauses). The create and destroy MVs module analyses the collected queries, determines which are suitable for conversion to use materialised views, applies any secondary parsing (for example when converting parameterised/prepared queries), prepares and executes the DDL queries to create the materialised views, maps the parsed queries to their originals and to the MV, and drops any unused or invalid MVs. The analyse query/MV use metadata module analyses the resultant mapped queries, analyses system metadata from the system views and plan cache, executes mapped queries, computes efficiency and efficiency ratios between the original and the mapped queries. The temporary tables reflect some of the data from the plan cache and store all the information needed by the described processes to operate.

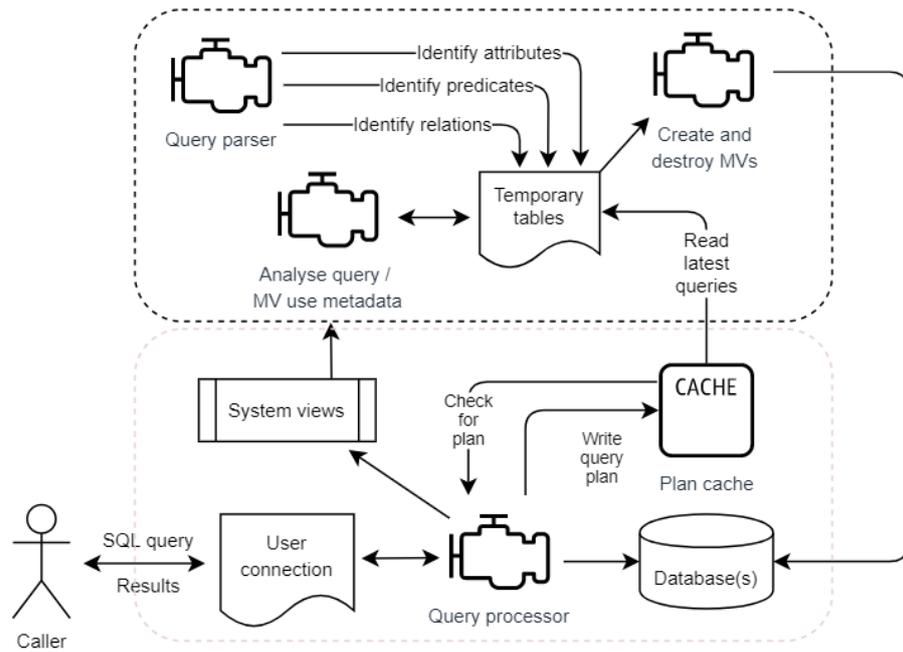


Fig. 8.1: High-level overview of the dynamic schema redefinition process

The data flow diagram in Fig. 8.2 illustrates the flow of data to, from and through the new components. This diagram uses Yourdon-DeMarco notation [3]:

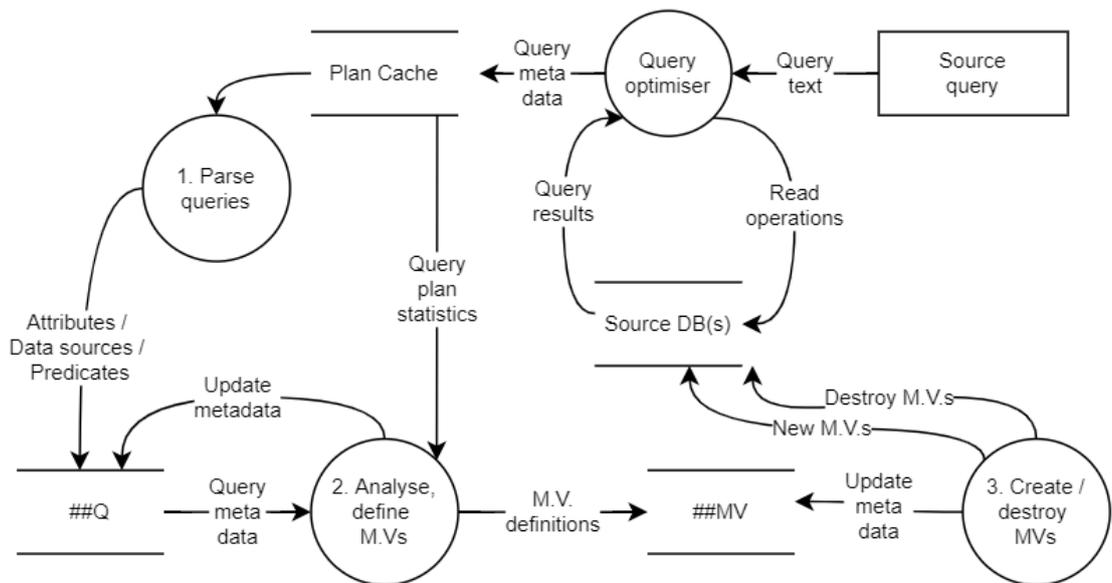


Fig. 8.2: DFD illustrating data flow in the dynamic schema solution

The query parser component is responsible for fetching and parsing queries from the RDBMS plan cache. The query is first tokenised, the existence of the relationships is mapped between the components of the query then each component is classified as either a data source (relational part), attribute (and the associated relational part) or predicate (clause on the WHERE or JOIN components). The use of these query items is then recorded by way of inserting new records to the query components temporary table or matching on existing components within the table and updating the frequency of the components' occurrence.

The algorithm for the query parser is shown in Appendix E, section E.1.

The create/destroy MVs component is responsible for a) identifying, through frequency analysis, the relational parts, attributes and predicates most commonly called and for constructing and implementing appropriate materialised views in the database; b) identifying those materialised views that are no longer required most frequently by inbound queries and destroying them. Improvements can be made in future by replacing or augmenting frequency analysis with total read count from parsing of the execution plan:

The algorithm for the create/destroy MVs component is shown in Appendix E, section E.3.

The analyse query/use MV metadata component is responsible for using the materialised view definitions created by the 'Create and destroy MVs' component to model queries from the plan cache, using appropriate system metadata, and to record the relative costs associated with running these queries against MVs versus the base schemata. The information output is stored within the temporary tables for use when creating/destroying MVs and for analysis. In a full implementation (where the query processor is exposed for re-engineering), this component would also be responsible for flagging the query to the query processor as suitable for running against the MV(s) and forcing it as an alternative rather than the base schemata.

The algorithm for the analyse query/use MV metadata component is shown in Appendix E, section E.4.

The temporary tables component is a set of tables held within the temporary tablespace/database of the RDBMS which enable the main processes of the schema redefinition process to read and write query, MV and performance data. These are recreated on system start-up/restart. As a static object, there is no associated algorithm; the entity-relationship diagram in Fig. 8.3 illustrates the structure of and relationships between the tables used for query analysis and the RDBMS-provided plan cache and supplementary table-valued functions. Crow's-Foot notation [4] is used and for the plan cache, it is assumed the structure provided for in the relevant Microsoft SQL Server 2017 plan cache table [5].

The `##cs` table, which is a reflection of `sys.dm_exec_query_plan` and derivation of the `sys.dm_exec_query_text` TVF, is not shown.

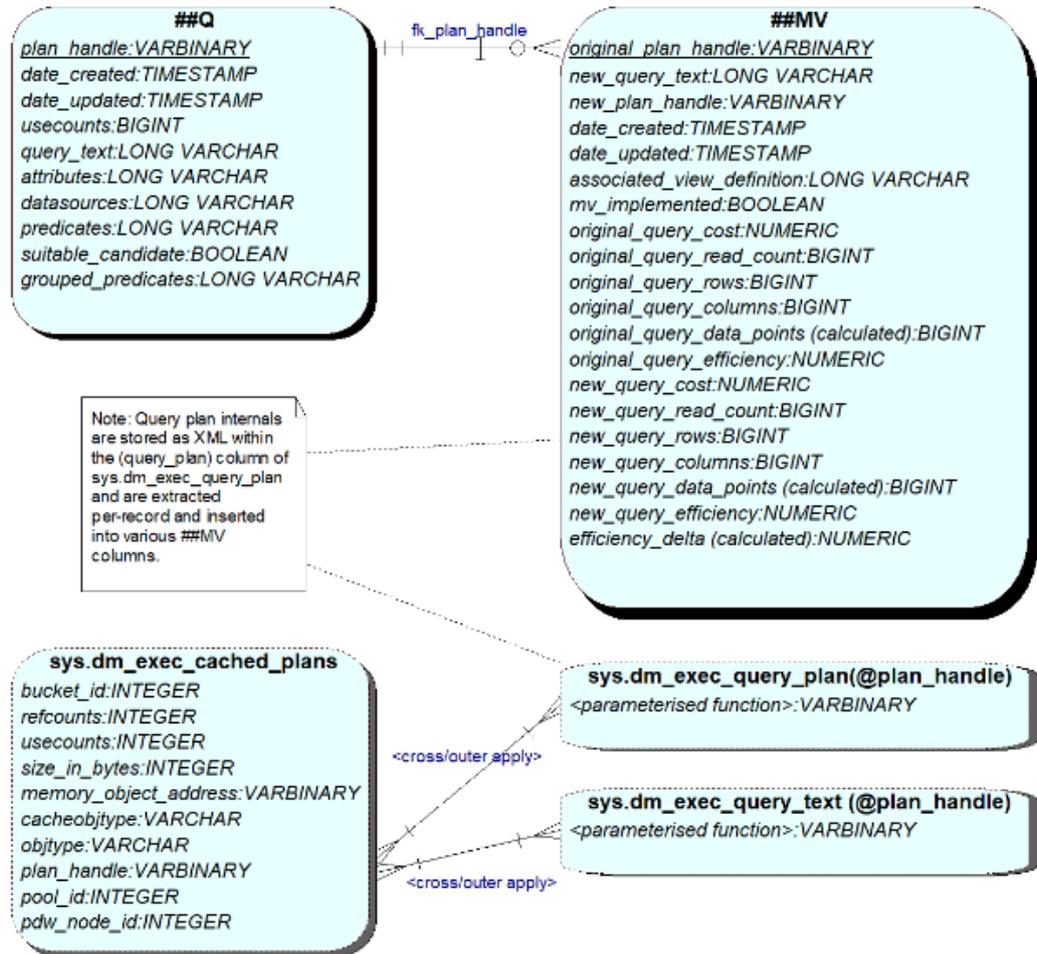


Fig. 8.3: ERD for tables involved in dynamic schema redefinition

8.3 Practical Implementation

To test this solution, the TPC-C benchmark data set [6] with the TPC-affiliated HammerDB open-source tooling [7] was chosen and Microsoft SQL Server 2017 Developer Edition was used as the RDBMS.

The process begins with zero materialised views and clearance of the plan cache. The algorithms from Appendix E, implemented in T-SQL, are then employed – these algorithms monitor the SQL Server plan cache, analyse query contents and periodically maintain a rank-order summary of appropriate subsets of the schema within global temporary table objects. The SQL Agent job

engine is used to operate this process. At set periods, each subset is constructed and implemented asynchronously as a materialised view, and views which are no longer highly ranked are dropped.

The HammerDB query simulation engine is not used out-of-the-box, since this runs just 5 different stored procedures with various parameters; suitable for performance testing, but unsuitable for analysing the impact of the algorithm due to a) the lack of variety in queries and b) the limitations of extracting statement-level queries from the plan cache, where procedures are aggregated as executions at the procedural level rather than at the statement level. Instead, a data bank of 9,660 different SELECT queries is used, generated from the TPC-C dataset by randomly selecting attributes from tables with and without different kinds of JOINS and using a datatype-appropriate random generation of predicates. These queries are stored in a separate database and separately executed using a Python script which allows the selection of a random query, the execution of the query and the control of the rate of execution using an artificial delay. All queries are unique with an estimated 4.48×10^{15} possible permutations across the schema (using formula $nPr = n! / (n - r)!$, assuming 94 columns and a mean average of 8 selections) ensuring the probability of exact query duplication from the process $p = 10,000 / 4.48 \times 10^{15} = 2.28 \times 10^{-12}$. Multiple processes can be spawned to simulate parallel users if necessary.

Query parser: This yields table `##q` populated with current and historical queries, parsed into SELECT, FROM and WHERE segments at the top hierarchical level.

Create and destroy MVs: This process was tested and execution was found to be consistently $10s \pm 5s$ (excluding the creation of indexes), with success rates in identifying and parsing queries, creating and testing MVs and linking queries to MVs with a typically c.68% success rate overall, +/- 10%.

Analyse MVs / use metadata: This module is responsible for analysing the created MVs, extracting the query metadata from the plan cache. This is done by extracting the query execution plans from the cache, parsing these plans by exploding the XML and extracting the key statistics, before updating the `##q_mv_link` table with the query statistics. This module also runs the new query versions generated against the MVs and extracting the new costs from the cache, updating the temporary table, to allow comparison of old and new query performance statistics on a per-MV basis. Finally, this module calculates the query cost and efficiency differences (the new definition of efficiency is used with the assumption that the appropriate 'rows read' parameter in the execution plan is accurate).

Temporary Tables: The temporary tables are of the global temporary type (prefixed ##), resident in memory, faulting to the tempdb database, and available across all sessions/connections in the database instance. This was chosen to maximise memory use, minimise disk use, improving performance, and avoiding metadata permanence.

Using these new processes, the MVs can be defined but in the current experimental configuration, redirecting the query engine to replace the base tables referenced in inbound queries with the MVs as they run is not possible, as this functionality is currently unavailable in all RDBMS engines, a drawback shared with other academic research in this area, although forks of open-source RDBMSs such as PostgreSQL could theoretically be developed to support this. The proposed solution allows for this by designating a component to perform this mapping and flagging operation. In lieu of this active replacement of queries, the same queries are re-run (while the TPC-C test load is in progress) that would have their references replaced against the new MVs and the number of reads required and query execution times against the original versions are compared in order to quantify any improvements in efficiency, drawing this information from the execution plans, which yields comparative statistics between the original queries and new queries. This also allows the computation and comparison of performance statistics.

8.4 Experimental Design

The implementation of the algorithms was completed in an iterative fashion and component-level (preliminary) testing was conducted throughout, combined with end-to-end (system) testing against the benchmark data set.

This section is structured into a set of preliminary observations (observations made on the success or limitations of the proposed solution found during implementation); and details of the test parameters, data and outcomes as systemic observations (observations made during end-to-end testing of the implementation); the results when considering storage and performance trade-offs are also presented, tangential to the main results.

After implementing solutions from the outcomes from the preliminary testing, the first successful system test was conducted. 1,801 queries were executed in serial over a maximum period of 300 seconds (actual: c.27 seconds) randomly drawn from the preassembled query bank of 9,660 queries that were created against the TPC-C benchmark data set. The query parser module was run, which parsed the resulting 1,464 queries associated with plans in the plan cache in approximately 2 seconds and identified 1,462 distinct queries in the raw query table ##q suitable for consideration for new MVs, a success rate of 99.9%.

The create/destroy MVs module was then executed, which identified 73 distinct non-indexed MVs to create, linking 1,186 queries in ##q to the new MVs in ##mv, a success rate of 65.6%. From the 73 MV definitions, 53 actual non-indexed MVs were created from the definitions in ##mv, a success rate of 72.6%. However, non-indexed MVs are only views, and only create an overlay which allows the query optimiser to access the base tables. For improved performance, indexed MVs are required. This module therefore then attempts to create the unique indexes on the identified MVs. It was found that of the 53 non-indexed MVs created, 6 were indexed successfully, taking a total of 910 seconds to complete the whole module run. The most common error encountered when indexing the MVs was due to the presence of OUTER JOINS (see below), the second most common was the presence of a duplicate primary key, and the third most common was the resulting row length of the MV exceeding the platform hard limit of 8,060 bytes. Thus, the overall success rate from inbound query to indexed MV was 11.3%. Tables were created in lieu of indexes as per the comments from the preliminary testing, which resulted in 100% of defined views having indexes or replaced with tables.

The analyse/use metadata module was executed to fetch costs and re-run alternative queries (new formulations of existing queries using the new MVs). This resulted in the successful analysis of 332 queries and fetching of relative costs for 171 total queries.

For these 171 queries, the following observations were made (using mean averages):

- 8 queries had increased in actual query costs.
- 33 queries had decreased in actual query costs.
- 130 queries had remained identical in actual query costs.
- The average cost increase using the new queries was 0.7622 in real terms.
- Estimated rows read increased, on average, by 1,261.
- Overall query efficiency, as per the definition (see 'Investigating Query Efficiency'), decreased by an average of 4.31%

Following analysis, outliers were found in the data that skewed the mean averages significantly. For example, cost delta (difference between original query and new query costs) averaged 0.76 but had a standard deviation of 11,005. This means that the mean averages are not representative of the data, and so the aggregated observations were repeated using *median* averages, which were all nil. Given that averaging had not proven particularly useful, outliers were excluded, and the data was analysed between percentiles 0.05 - 0.95. The resulting data offers some evidence that queries against MVs (or tables in their place) can, on count of affected queries alone, be beneficial to more queries than harmful to others and on balance may indicate some limited evidence of viability. The tests were continued by examining those queries benefiting from the new arrangements, and the expected cost and read count/rows (efficiency) effects.

8.5 Testing and Results

The solution was successfully implemented and tested this using Python as the application caller and Microsoft SQL Server 2017 as the database engine. The code is listed in Appendix E.

8.5.1 Preliminary observations

The following observations were made during implementation and testing:

- Due to the propensity of queries involving OUTER JOINS to be susceptible to missing rows when base data is added, OUTER JOINS are not supported in indexed views in Microsoft SQL Server [8]. A similar, but less severe, limitation exists in Oracle Database [9]. When it was found that the overall success rate in creating unique indexes on the views was relatively low, the views were stubbed by creating tables instead with a small code change. Although tables are not schema-bound (meaning when the base table is updated, the tables are updated), they provide an appropriate facsimile for MV testing. This does, however, impact the overall solution strategy. The implications of this are discussed further in the conclusions.
- Time taken to create indexed views (where allowed) was relatively long and in some cases indefinite. One of the symptoms of poor performance was identified as CXPACKET-type waits, due to the CREATE INDEX statements using parallelism inefficiently (8 logical cores were available on the test system but one node alone was used for the I/O operations and another to govern) and coupled with the large requirement for row reads while creating these indexes on views with large row counts. Negligible actual disk use was observed throughout and highly variable (c. 12.5% to 100.0%) CPU use was noted. Microsoft SQL Server supports parallel index operations on the CREATE INDEX statement [10] but this wasn't working very successfully (over 300 parallel threads were observed on a single core in the preliminary testing with most other cores idle). The lack of clustered indexes on the TPC-C data set was noted and, in lieu of this, and to help address performance, a set of database statistics covering each key on each table in this set were created. Unique clustered indexes or defined primary keys to lower I/O could not be used due to the limitations of the data (particularly, key duplication) in the TPC-C dataset supplied by HammerDB.

- It was also noted that abortive runs of the process to create the indexed views (where the process hung) occurred when a CROSS JOIN was involved. Given a CROSS JOIN outputs the Cartesian product of two relations, the number of rows involved can be tremendous before the WHERE filters are applied. On analysis of one occasion where the process hung, it was noted that the CROSS JOIN of STOCK and ITEM with 5 unindexed primitive predicate filters resulted in 10 billion rows to store in memory, pre-filtering, and consequently 1×10^{10} rows in the destination index table, too many to store efficiently or be of any use to subsequent queries; creating this index also resulted in 100% CPU use. For this reason, the process was amended to exclude the consideration of queries that contain CROSS JOINS.
- It was noted that queries with excessive numbers of predicate filters took longer to run due to the excessive filtering required on the table/index scans which could affect scalability of this solution especially for tables larger than 100,000 row cardinality.
- It was noted that the new process successfully deduplicated queries, deduplicated their predicates and attributes, deduplicated and ignore repeated MV definitions and consequently ran continuously against a consistent inbound query load.
- Some indexes could not be created due to the space required to store the output index, and the test system used wasn't sufficiently powerful to process queries joining the columns in order_line (originally >1.4m rows) in a reasonable time period when a table scan was used. This was particularly prevalent when merge-type INNER JOINS were used with parallelism due to the requirement for merge-type JOINS to have pre-sorted input, and the computational load this entails with very large data sets. To counter this, 50% of the content in order_line was removed at random to reduce the order_line cardinality to 721,198, which did not affect database integrity since there are no foreign key dependencies (hard or soft) upon this table.
- Related to the above, as queries are automatically generated, some query joins are illogical, and effectively create CROSS JOINS. For example, the join between CUSTOMER and HISTORY on `c_w_id = h_c_w_id` is illogical since the columns relate to WAREHOUSE, and for all rows in both tables, all values are 1. With 100,000 rows on each side of the JOIN, this amounts to a CROSS JOIN that quickly fills available disk space. This is an artefact of the artificiality of the queries. Working around this issue by imposing a TOP 1,000,000 clause within every indexed view so there exists consistency, if not cardinality, between the query executed and the MV created would be appropriate to reduce output

rows, except that indexed views may not have row count restrictions since they become non-deterministic. Instead, the row counts were calculated and each of the 10,809 queries originally generated were executed using a Python script, and those queries were removed that took longer than 20 seconds to execute from the query bank or that would result in more than 1m rows returned, leaving 9,660 queries available for testing, therefore eliminating 1,149 problem queries that could crash the index creation process further along the process.

- It was found that the initial existence of more queries in the plan cache than were executed anomalous (the plan cache and buffers were cleared before testing). Analysis of the plans in the cache revealed the cause to be the existence of the queries used to select individual queries from `tpcc_queries` to run, in the form `SELECT query FROM tpcc_queries.dbo.queries WHERE id = N`. This query was not parameterised in Python and so iterations of `N` occupied the plan cache. This was filtered out in the diagnostic and measurement by excluding queries containing the `'tpcc_queries'` string, by adding a plan guide forcing parameterisation and by parameterising this in the `pyodbc.cursor.execute()` call in Python. No difference is made to the test output since this query is assessed, deduplicated and rejected by these processes as not belonging to the database under test (`tpcc`) and through explicit filtering later in the process.
- It was found that the plan cache management by the RDBMS was unpredictable. Plans are stored in virtual 'buckets' which are stored within the cache. Each type of plan is allocated to a bucket. There is both a maximum number of items per bucket (for SQL Server, in the region of 160,000) and a maximum number of overall plans allowed in the cache, tempered with a maximum overall cache size allowed. Microsoft SQL Server is one of the few mainstream RDBMS systems that does not allow direct control over the size of the plan cache – hence, the only adjustment possible is that of maximum server memory allocated to the instance, which was set to 14GB. Of this, upon reading the system documentation it was indicated that 75% of the first 4GB is allocated to the plan cache (3GB) plus 10% of the remainder (10GB) as maximums. It was found, however, that the plan cache would periodically flush on or around 2,000 plans regardless of the maximum memory setting. This doesn't present an operational issue for creating/destroying MVs since the query parser is run frequently to capture and store the necessary information to create/destroy the MVs, but it does present issues analysing the plan cache to extract original query metadata if the cache has been flushed before this step is applied.

There was no workaround to this issue by creating a ‘mock cache’ since the metadata extraction relies on table-valued system functions and views, the population of which over which there is no control. In lieu of this, two actions were taken: a) the modification of the Python caller to parameterise the call to fetch a random query, saving around 25% of the plan cache; b) the maximisation of the memory available to SQL Server; and c) desisting from running the client-side Profiler tool alongside the testing, relying on simple loop counts instead, since it was theorised the memory required to store the Profiler data was being reallocated from SQL Server by the operating system, with the plan cache the first casualty. Although turning on the ‘optimize for ad hoc workloads’ setting was considered, which would leave stubs for single-use plans in the cache rather than full plans, it was found this would not be suitable for plan performance metadata extraction. Finally, after some trial and error, a ‘hard stop’ of 1,800 successful executions was put in the Python caller to maximise cache population before flush.

- To ensure that the query runtime statistics were available before cache flush and unaffected by the subsequent process code, these statistics were dumped into the ##cs table immediately prior to query parsing and the Analyse/Use Metadata module was amended to use this table rather than the system views.

8.5.2 System testing

5 successful end-to-end system tests were completed and the data compiled from each of the stages into the summaries and the series of graphs shown in the following Tables 8.4 – 8.7 and Fig. 8.8. With 1,800 queries run from a bank of 9,960 available queries, given random distribution, it is surmised that 5 tests are sufficient to cover a reasonable proportion, approximately 64% of the queries available.

Table 8.4: Query Parser / Create/Destroy MVs Phase – Summary Metrics

| Test # | Queries Executed | Plans | | MV Links | | MVs Defined / Created | | | Indexes / Tables Created | | Queries with valid MVs | |
|--------|------------------|----------|--------|-------------|-----------|-----------------------|----|-------|--------------------------|----|------------------------|-------|
| | | Cached / | Parsed | Created (Qs | with MVs) | | | | | | | |
| 1 | 1801 | 1464 | 1462 | 1186 | 65.6% | 73 | 53 | 72.6% | 47 | 6 | 1186 | 65.6% |
| 2 | 1801 | 1298 | 1287 | 1019 | 56.5% | 74 | 47 | 63.5% | 10 | 35 | 1019 | 56.5% |
| 3 | 1801 | 1283 | 1272 | 1032 | 57.3% | 74 | 52 | 70.3% | 11 | 38 | 1032 | 57.3% |
| 4 | 1801 | 1261 | 1251 | 991 | 55.0% | 74 | 43 | 58.1% | 10 | 30 | 991 | 55.0% |
| 5 | 1801 | 1250 | 1240 | 994 | 55.2% | 73 | 48 | 65.8% | 12 | 34 | 994 | 55.2% |

Table 8.5: Analyse MVs/Use Metadata Phase, Summary Metrics I

| Test # | Original Query Metadata Captured | New Query Metadata Captured | Comparable Query Metadata Captured | # of New Queries with Lower Costs than Original | # of Original Queries with Lower Costs than New |
|--------|----------------------------------|-----------------------------|------------------------------------|---|---|
| 1 | 332 | 171 | 171 | 33 | 8 |
| 2 | 301 | 166 | 166 | 32 | 17 |
| 3 | 311 | 153 | 153 | 27 | 23 |
| 4 | 268 | 166 | 166 | 37 | 19 |
| 5 | 288 | 169 | 169 | 42 | 24 |

Table 8.6: Analyse MVs / Use Metadata Phase, Summary Metrics II

| Test # | Avg. Cost Differential for All Original vs. New Queries | Avg. Read Count Differential Between Query Pairs | Total Read Count Differential for All Query Pairs | Average Efficiency Differential Per Query Pair | Total Cost Differential For All Query Pairs |
|--------|---|--|---|--|---|
| 1 | 0.7622 | 1261 | 215653 | -4.31269 | 129.99 |
| 2 | 2.86199 | 3937 | 653536 | -5.82934 | 475.10 |
| 3 | 2.69643 | 3888 | 594922 | -2.18379 | 412.55 |
| 4 | 2.60160 | 3641 | 604448 | -7.06265 | 431.86 |
| 5 | 5.66083 | 7809 | 1319851 | -4.67521 | 956.68 |

Table 8.7: All Phases – Storage and Runtime Costs – Summary Metrics

| Test # | All Query Executions (S) | Query Parser (S) | Create/Destroy MVs (S) | Analyse/use Metadata (S) | Total Runtime (S) | Total Additional DB Objects (MB) |
|--------|--------------------------|------------------|------------------------|--------------------------|-------------------|----------------------------------|
| 1 | 27 | 2 | 910 | 27 | 966 | 8,211 |
| 2 | 26 | 2 | 354 | 66 | 448 | 5,070 |
| 3 | 27 | 2 | 2400 | 83 | 2,512 | 11,521 |
| 4 | 27 | 2 | 842 | 42 | 913 | 7,795 |
| 5 | 26 | 1 | 629 | 63 | 720 | 13,528 |

Differentials were captured for many key metrics, as detailed in Tables 8.4 through Table 8.7. Fig. 8.8 shows the total cost differences between the original query versions and the new query versions, expressed as a floating-point number. Cost differences greater than 0 indicate the new queries consumed more system resources (as a blended measure as defined by query cost in SQL Server) than the original queries.

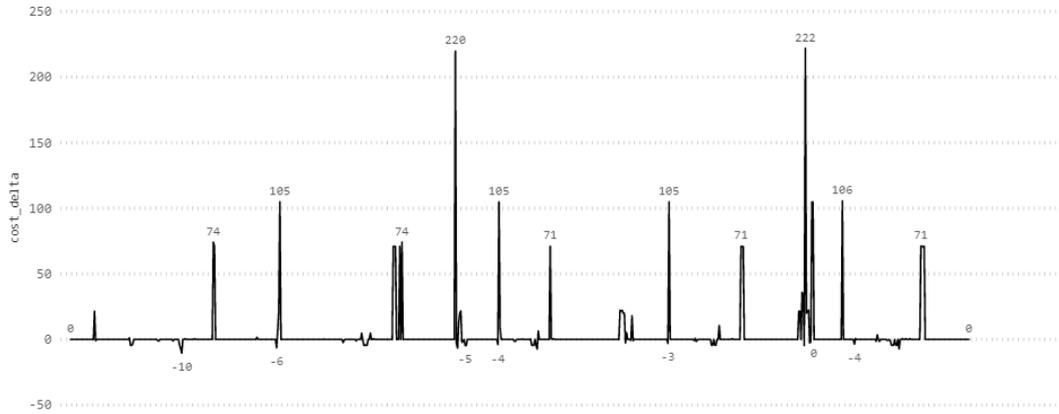


Fig. 8.8: Cost deltas for all queries, all runs

However, it is noted that the standard deviation is somewhat high, and that the mean average of this data set does not necessarily correspond to the average cost impact of new queries using MVs that original queries. In Table 8.9, the number of original queries that cost more than the new queries are counted, likewise the vice versa case, and some basic metrics on the whole data set are captured.

Table 8.9: Metrics for the cost delta measure, all queries, all runs

| | |
|--|--------|
| Cost delta: Mean average | 2.92 |
| Cost delta: Standard deviation | 17.19 |
| Cost delta: Original queries > new queries | 171.00 |
| Cost delta: New queries > original queries | 91.00 |
| Cost delta: New queries = original queries | 563.00 |

This yields the result that there were 171 cases of 825 (20.7%) where the newly generated queries outperformed the original queries, versus 91 cases (11.0%) where the opposite held true (and 563 queries (68.2%) with no difference observed). In other words, there were more queries that had improved performance by the new MV process than queries that were adversely affected.

With this in mind, the data is filtered in Fig. 8.10 to observe the cost deltas for all queries where the new cost is lower than the original cost, to observe the amplitude overall.

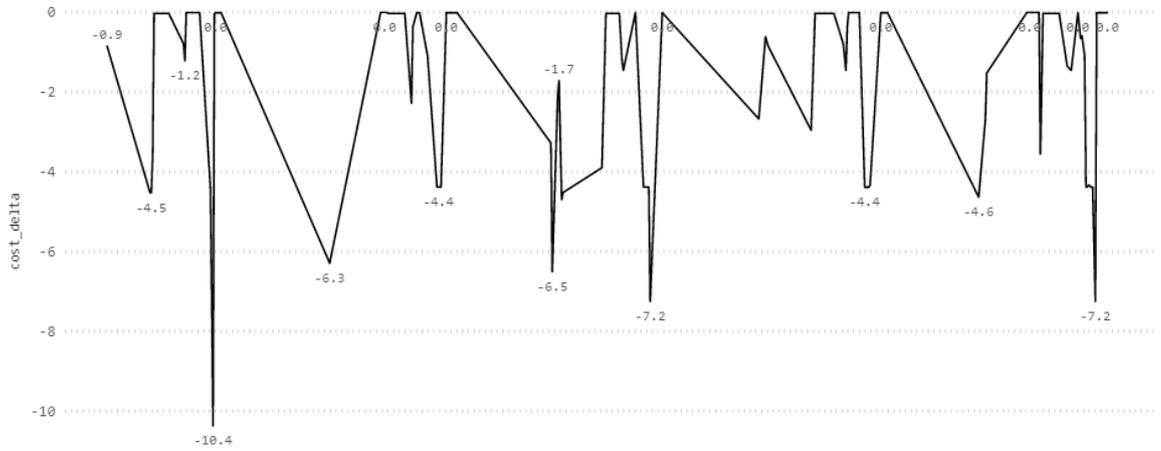


Fig. 8.10: Cost Deltas where New Query Cost < Original Query Cost, all runs

Table 8.11: Metrics for the cost delta measure where new query cost < original query cost, all runs

| | |
|--------------------------------|-------|
| Cost delta: Mean average | -1.15 |
| Cost delta: Standard deviation | 2.00 |

The cost savings range from nil to -10.4, with a mean average of -1.15 cost saving and a reliable standard deviation of 2.0 indicating stability, as shown in Table 8.11. However, as cost in SQL Server is somewhat relatively defined, this figure needs to be placed in context, particularly in reference to the average query cost. The analysis above was repeated for the other two key measures captured, row reads required and efficiency. Tables 8.12 and 8.13 show the total increases (for original query < new query), total savings (for original query > new query), and averages for all measures, for cost, rows read and efficiency. The second table shows figures excluding extreme outliers in the 0-5th and 95th-100th percentile, based on cost delta, necessary due to the high standard deviation in the data.

Table 8.12: Performance metrics for all processed queries, full range.

| | <i>(filtered based on query cost)</i> | | | | Original | New |
|-----------------------------|---------------------------------------|----------------|----------------------|------------------|----------|----------|
| | Original > New | Original < New | All (inc. identical) | | | |
| Total Cost Difference | -196.62 | 2602.8 | 2406.19 | Total Costs | 663.84 | 3070.03 |
| Avg. Cost Difference | -1.15 | 28.6 | 2.92 | Avg. Costs | 0.805 | 3.72 |
| Total Read Count Difference | -159307 | 3547837 | 3388530 | Total Read Count | 670345 | 4058875 |
| Avg. Read Count Difference | -931.62 | 38987.22 | 4107.31 | Avg. Read Count | 812.54 | 4919.84 |
| Total Efficiency Difference | -1220.24 | -2781.53 | -4001.77 | Total Efficiency | 44000.92 | 39999.15 |
| Avg. Efficiency Difference | -7.14 | -30.57 | -4.85 | Avg. Efficiency | 53.33 | 48.48 |

Table 8.13: Performance metrics for processed queries, filtered between 5th - 95th percentile on cost delta.

| | <i>(filtered based on query cost)</i> | | | | Original | New |
|-----------------------------|---------------------------------------|----------------|----------------------|------------------|----------|----------|
| | Original > New | Original < New | All (inc. identical) | | | |
| Total Cost Difference | -27.47 | 75.12 | 47.64 | Total Costs | 303.21 | 350.85 |
| Avg. Cost Difference | -0.2 | 1.47 | 0.1 | Avg. Costs | 0.4 | 0.46 |
| Total Read Count Difference | -5522 | 114876 | 109354 | Total Read Count | 294893 | 404247 |
| Avg. Read Count Difference | -40.92 | 2252.47 | 146 | Avg. Read Count | 393.72 | 539.72 |
| Total Efficiency Difference | -1224.85 | -2131.13 | -3355.98 | Total Efficiency | 43069.43 | 39713.45 |
| Avg. Efficiency Difference | -9.073 | -41.79 | -4.48 | Avg. Efficiency | 57.5 | 53.02 |

An average cost saving of -0.2 excluding outliers was noted against an average cost of 0.4 (50% cost saving) in cases where the original query has higher query costs than the new query. When including the vice versa case and cases where query costs are identical, a modest increase in costs (0.46 - 0.4 = 0.06) is observed, a 15% increase overall. Checking the read count totals and averages, a decrease in number of read counts executed is observed, -5,522 in total with an average read count saving per query of -41 reads. Overall, a significant increase in read counts from 294,893 (original queries) to 404,247 (new queries), was observed: 37.0%. For cases where new queries outperformed original queries, a drop in efficiency was noted, by an average of 9%; for the vice versa case, the drop in efficiency by the new query was pronounced at -41.8%, reflecting the change in the read counts required.

It was found that in approximately 15-25% of queries, query costs dropped after implementing the new MVs. This result was borne out across all 5 test groups. This drop shows applicability of the new MV process to some queries in the set.

8.5.3 Storage and write performance trade-offs

During experimentation the total size of new database objects in the database (views and tables, the latter masquerading as views) was recorded. Table 8.14 summarises these findings.

Table 8.14: Storage required by new materialised views

| Test # | Total Additional DB Objects (#) | Total Additional DB Objects (MB) | Avg. MB Required / New MV |
|--------|---------------------------------|----------------------------------|---------------------------|
| 1 | 53 | 8,211 | 155 |
| 2 | 45 | 5,070 | 113 |
| 3 | 49 | 11,521 | 235 |
| 4 | 40 | 7,795 | 195 |
| 5 | 46 | 13,528 | 294 |

It was found the use of MVs to simulate subsets of schemas is viable if and only if the provision of additional disk space to support this strategy is provided. An average of between 155MB and 294MB was required per MV, with a consistent average of between 40 and 53 new MVs successfully created per test (1,800 inbound queries). This implementation removes MVs that are no longer used; it also deduplicates the query and MV definitions, shown during preliminary testing when the number of MVs did not significantly increase during multiple executions of the process modules given a steady inbound query load.

8.6 Conclusions

This chapter began with the observation that querying large tables for small quantities of data is inherently inefficient and a simple efficiency metric was proposed comparing reads required against available rows of data, defining this for both straightforward contiguous page reads in a table and against a B+ tree index structure, such as used for clustered and non-clustered indexes. It was noted that for all cases except row lookups and single-row index seeks, efficiency remained less than 100% and the intention was expressed to improve the overall efficiency of queries through the use of materialised views to supplement base tables; where such an approach has been identified before in the literature, the solution has been extended to create a set of processes that analyse inbound queries, create appropriate materialised views to support those queries, and remove materialised views that no longer meet the requirements of the inbound query stream. Two key performance metrics were focused upon, the second being a function of the efficiency; query cost, a blended measure of CPU, I/O and expected work provided by the query optimiser, and read counts, which were extracted directly from the RDBMS plan cache.

The results showed that against the TPC-C benchmark dataset, using a query data bank of 9,960 queries in 5 test sets of 1,800 randomly-drawn queries, it was possible to demonstrate some benefits of the new process. These included a modest reduction in the average plan cost by 50% for queries which demonstrably ran at lower cost against the original query; however, when including all queries, including those where MVs made no demonstrable difference (or indeed worsened the

difference), average query cost rose by 15%. Similarly, a minor read count drop (c. 1%) was demonstrable for queries positively affected by the new process, but an overall increase in reads of 37% when considering queries with original costs above new query costs. Consequently efficiency, as a function of reads, showed significant drops – approximately 9% and 41% respectively.

It was found that, queries which would not benefit from the MV approach notwithstanding, the new process has produced significant evidence of plan cost savings approximating 50% (average 1.15 drop against a landscape of queries ranging in relative costs from 0.8 – 3.72) in approximately 15-25% of cases, meaning that 15-25% of queries could benefit from these views by a reduction in cost varying up to 50%. This means the proposed solution, while in its current form does not provide evidence of viability to *every* presented query, may be effective if highly targeted towards *certain* types of query. The evidence amassed suggests that given improvements in the implementation, this solution could achieve the following:

- By including parsing support for parameterised queries, the improvement of the query uptake into the new dynamic query process by up to 30% (using the same TPC-C implementation), this figure based on the exclusion percentages noted during testing;
- By improving parsing overall, the exclusion of queries which are unsuitable for the new process – for example, those involving side-effecting functions – earlier in the process.
- By analysing indexes required by the queries, the creation of effective, targeted materialised views indexed for the queries using them, significantly lowering read counts.

Some difficulties were found that are inherent to the RDBMS engine. Materialised views in Microsoft SQL Server do not support OUTER JOINS since the addition, update or deletion from a base table in an OUTER JOIN can result in the removal of a row, inconsistent with a schema-bound database object. Oracle Database has some workarounds to this issue but also presents some weaker limitations. It is suggested that the implementation could be extended by either removing queries with OUTER JOINS from scope or creating multiple views for the components of the JOINS and assigning an MV to each component, retaining the OUTER JOIN in the original query. This approach would also be compliant with relational theory, reducing each one-to-many relationship to a one-to-one relationship. As this solution was out of scope, this issue was overcome through replacement of the MVs with tables which have the advantage of identical structures to MVs in this context but the disadvantage of being static.

Some RDBMSs may not be suitable candidates for a process reliant on the plan cache. As of MySQL 8.0, the plan cache has been removed from this RDBMS, citing implementation difficulties

(Lord, 2017). Some inventiveness may be required to overcome this consideration – for example by fetching plans manually from the query optimiser and storing as a task to be done by the database administrator. However, all other major RDBMS platforms retain their plan caches. More work is required to overcome the periodic flush of the plan cache noted during testing and which appears to be unique to the Microsoft SQL Server RDBMS; in many other RDBMS systems, the plan cache is directly configurable.

Future work on this project would include addressing the implementation points above; more testing for asynchronicity and long-term effects; improvement of the query parser, perhaps integrating industry-standard parsing tools such as GNU Bison; extending the implementation of the MV definition to include queries with OUTER JOINS and more complex structures such as subqueries, CTEs, system functions, TVFs and variables; extending the implementation with an indexing strategy to index or re-index MVs periodically in response to query process flow; and further theoretical work on the efficiency of the various access components in query execution plans versus theoretical maximum efficiency, to strive to achieve the maximum possible efficiency for data access in relational systems.

8.7 Chapter Summary

In this chapter, it was demonstrated how the theoretical design for dynamic schema redefinition as part of the PETAS framework can be illustrated through a set of algorithms and implemented using standard SQL and simple Python routines. Each component of the dynamic schema redefinition process was implemented, and these were tested independently, adjusting the implementation in response to the observations made through unit testing, then testing the components end-to-end. Modest improvements were found in query performance under some limited conditions; it was evident that further work to improve the range of queries to which the new solution could apply is desirable; likewise, the implementation workaround using materialised views, rather than direct logical-to-physical mappings (e.g., from new schema to page offset), was necessitated by the need to rewrite some large part of the query engine. It was found the restrictions inherent in materialised views, including the restriction on outer join operations, contributed to the low query improvement metrics, and a better, more robust implementation, particularly of the query parser, may have effected substantial improvements. This latter finding mirrors one of the key conclusions from the query parser and schema selection mechanisms detailed in earlier chapters.

However, the presence of positive improvements as measured through both ordinary metrics and the new efficiency definition (function of rows read vs. rows available) is encouraging, and it is found that despite open questions about the effectiveness of the implementation, the key ideas

remain viable and further work can focus upon improving, refining and further validating the solution.

In the final chapter, the results and conclusions are gathered into a narrative, and the findings are compared and contrasted with the stated aims and objectives. It is sought to conclude whether the research has been successful in this regard and whether a novel contribution to knowledge has been made. Chapter 9 concludes with a summary of potential research directions and unresolved questions to investigate to support the future of PETAS.

Chapter 9: Conclusions and Future Work

9.1 Introduction

This final chapter reflects on the research completed and summarises all the findings. The outputs are drawn together, starting from the initial literature review, qualitative research and proof-of-concept problem investigation and leading into the main body of research, the development of PETAS, into a set of conclusions which are presented through mixed methods evaluation. The results are examined and validated against the research questions, aims and objectives, seeking to understand if, and to what extent, these have been successfully met. The novel contribution to knowledge is revisited, and next steps considered to develop and integrate this research into existing database platforms. A final summary and detail on the contents of the Appendices to this document is provided.

9.2 Problem Investigation

9.2.1 Qualitative research

Following the literature review, the design, piloting and deployment of a questionnaire was undertaken and aimed at data practitioners, with the objective to discover, through a mixture of structured, multi-choice questions and open questions, the current views of practitioners on database performance challenges and their potential causes. This survey was piloted with a small group of individuals including database experts, then the questions and structure were amended in response to their feedback and the survey was delivered to a wider audience.

Given the initial findings from the literature review weighed against the popularity and ongoing suitability of relational databases, particularly as solutions for object-oriented data and increased volumes of data, particular difficulty was found in structuring questions in an objective, unbiased format. Likert scales were employed as a tool to help reduce bias, adjusted questions in response to pilot feedback and internal consistency between questions was sought. A simple filter was also employed at the start of the questionnaire to pre-qualify respondents and details of their experience collected to further validate the data.

Using thematic analysis, four key themes were extracted; negative ORM behaviour, ORM use (prevalence thereof); education, awareness and perception; and future outlook. From here, it was established that many respondents had a poor view of ORMs but an equal number were not generally familiar with their use. There were few positive opinions for their role in query

performance tuning. It was found the respondents placed the majority of the blame for poor query performance within their organisations on a lack of education or awareness of relational database systems in others.

Due to the relatively low number of respondents, few definite quantitative conclusions can be drawn from the survey outcomes since the potential variance in replies is too high to infer any reliable statistical output. The high proportion of respondents not able to respond in depth on ORMs was reflective of a lack of experience with ORMs in general within the respondent population; in retrospect, not surprising since ORM frameworks sit within application development frameworks (e.g. Entity Framework within .NET; Hibernate within Java; Django ORM within Django) and as such, database administrators, developers and architects are unlikely to come into regular contact with them. On reflection, a more suitable target audience (or a combined audience) should have included application developers.

The questionnaire findings were supplemented with a small range of interviews with invited database practitioners. These were conducted as semi-structured interviews; with a range of ‘starter’ questions and points on which to follow up, aligned to each theme of the questionnaire findings, thus triangulating outcomes, and each interview took place over the course of an hour with the pace dictated by the participant; the flow of conversation was directed, but in general sought to elicit the detailed experience and opinions of each candidate in a contextual, meaningful way. The interview participants were found to be open and participative in the conversation with very little direction required by the interviewer; all participants were well-experienced and came from different sectors including consumer website provision and the aeronautics industry.

The findings were analysed using the NVivo software package which allowed codification, annotation and grouping of the transcript contents, reminiscent of grounded theory and the approach to the literature review. A frequency breakdown was produced of various survey topics and, although initially the intention was to perform a semantic analysis, it was found that the results of this were inconclusive due to the dialogue being mostly technical in nature and not expressing many marker keywords. Instead, a narrative analysis was used, described in detail in Chapter 4.

It was found that the interviewees generally held opinions compatible with the findings from the literature and the survey. Some were heavily critical of the speed and flexibility of relational platforms – one particularly striking quote:

“[Participant] I think NoSQL DBs will be the future thing. Databases with designs 50-60 years ago, yeah, the initial concepts. So for stuff that was applicable at the time it, it was good, but with the modern web applications and user interfaces, and just the volume and in different types of data we can collect. Trying to put it all into a SQL DB doesn't make sense when you can have something something like BigQuery or MongoDB, that you can store different types of stuff in there and install get good performance and usage.”

The general apathy and, in places, hostility towards the usefulness of relational database systems was apparent throughout all three strands of the qualitative research. Interview participants also attacked the lack of awareness in their peers in writing SQL queries (reflecting the awareness issues found in the questionnaire); the potential scalability of the systems to store data at volume; and the speed that relational database systems responded to real-time queries. There was little output from the interviews on ORMs.

In general, this qualitative research serves to reinforce the case that relational database systems are in a difficult position; able, theoretically, to support querying at scale, they are nonetheless struggling with the query antipatterns presented to them by ORM platforms. Relational database systems are perceived as slow and inflexible; a wide range of interventions have already been discovered and implemented in the last fifty years, and active research has slowed and has moved to nonstructured and alternative data representation forms.

9.2.2 Quantitative research

Following the qualitative research, it was sought to reproduce some of the findings of others and validate the opinions of the survey participants. In Chapter 4, section 4.5, the quantitative experiments are described. Two studies were undertaken; first, to see if the reported object-relational impedance mismatch query performance impacts from Ireland et al. [3] and others were reproducible in a modern relational database platform, and second, to compare and contrast auto-generated queries from an ORM layer against queries written manually, and to compare performance results. This latter case is set against a real-life data set, Pacific Ocean buoy sensor readings.

The first study found that for trivial queries, there was none, or little, difference between execution plans and consequently between performance outcomes. The ORM framework was able to deal with simple cases, to prepare queries appropriately, to parameterise and transpose parameters into stored procedure query calls and single SQL queries. However, as queries became more complex, especially for selections/projections with multiple selected columns and more than one join, the

elapsed time was much higher than manually-written queries (290ms vs. 118ms, for example, as shown in Chapter 4 with all other confounding variables controlled for).

It was found that ORM platforms tended to have mitigations for some of these performance defects which are not always used effectively; for example, the use of lazy loading over eager fetching can reduce the number of rows transmitted to the client; the forcing of JOINS over nested queries can simplify execution plans; and the proper parameterisation of literals can reduce recompilations through better matching in the query cache.

The new query representation solution was developed as a direct response to the latter weakness, considering queries as computable and relatively comparable objects rather than as strings to be parsed and bound.

In the second study, it was sought to replicate the emergence of anti-patterns claimed in the literature by generating queries against a real-life data set. The data set chosen is a set used before in the literature for data mining purposes and comprises more than 2m data points set over a single table with 178k rows. To eliminate the possibility that anti-patterns are confined to a particular database product, the Python and the Django framework with Django ORM was used against PostgreSQL for the investigation, versus the previous use of Microsoft SQL Server. It was sought to expand the range of measures from simple execution time to a range of standard metrics as described in Chapter 4. As the presented data was somewhat simple in schema structure, this was restructured as a Kimball-type data warehouse schema [4] using fact and dimension tables to have a consistent and industry-recognised data structure upon which to base the testing.

It was found, with one outlier result, that there was a positive correlation between the complexity of the presented database query and the time taken to execute the query. Corresponding metrics such as memory use and plan cache size also increased in line with the execution plan. To statistically validate the results, t-testing was conducted on the observations of the mean execution time across the range of tests, but the resultant p-values indicated an 18% chance that these results were due to chance alone, due in part to a low population of query tests which were analysed manually. However, direct comparison of figures yielded some observational evidence between sets of figures that performance differed, if unreliably so, between ORM and non-ORM-generated queries.

9.3 Query Representation

Following the problem validation and investigation phases, which encompassed both secondary research through a detailed literature review and primary research through the administration of survey instruments and the investigation of quantitative outputs using a positivist experimental

approach, the solution, PETAS, was then designed to introduce several features to address opportunities to improve relational database query performance in response to the findings that such performance is suffering under the burden of increased data flows, apathy from the development community for relational solutions and the practical effects of object-relational impedance mismatch.

PETAS consists of three key areas; the query representation alternative, incorporating a query parser and adjacency cube generator, rendering queries from SQL into multidimensional arrays; the schema selection mechanism, which compares the adjacency cubes of an incoming query, the cubes of all queries in the cache and computes similarity scores before selecting an appropriate schema variant for the incoming query by using K-nearest neighbour selection on the cached cubes to predict the most appropriate choice. The third area is the dynamic schema redefinition procedure, which runs asynchronously and is responsible for generating, destroying and assessing the usefulness of variant schemas. These schema variants are used by the second component.

First, the query representation mechanism was built, which converts inbound SQL queries to multidimensional arrays, each array consisting of three dimensions and the intersection of each X-Y-Z co-ordinate marked with a binary value to indicate association between each dimension. Columns in the X and Y axes indicate bindable objects in the query and columns in the Z-dimension indicate subtype of relational expression, with 4 possible choices: selection, intersection, predication and membership each correlating to their set-theoretic alternative.

The construction of the theoretical model was relatively straightforward, since the language for proving that SQL queries are representable relationally and *vice versa* has been long established by Codd [5] and others. It was shown in the theoretical design that queries constructed as directed graphs, which are representable as adjacency matrices, have the characteristic of embodying relationships that the current query parsing methods do not. These relationships, when coupled with the objects in the query, have the effect of producing a mathematical structure which can be compared with other mathematical structures of the same type using measures such as Hamming or Manhattan distance. From this theoretical work, a series of equations, or transformative steps, were presented and expanded into a set of algorithms that demonstrated the process.

From the implementation perspective, a working implementation was created from the theoretical design using Python, ingesting SQL queries and outputting multidimensional lists. The experimental solution was validated by generating queries against another real-life data set used later in the research component concerning crime data in the US city of Chicago; a random query generator was written and used in Microsoft SQL Server against this data set to produce 1,000 queries, which were validated manually and it was found that 947 were valid and suitable for test purposes. The functional testing of these 947 queries showed 100% executed normally. A test harness was then constructed and the duration recorded for how long the query representation

algorithm took to process each query into the equivalent multidimensional array; these results were presented in Chapter 6 where it was found the mean average duration to be 1.8ms +/- 1.5ms standard deviation. This overhead is found to be minimal in the context of query execution which can take many seconds to resolve a given query and consequently this solution is believed to be viable.

In Chapter 7 the testing of the query representation mechanism was extended by coupling it with the similarity scoring mechanism and schema selection process, which are both reliant on the existence of the query representation process and running end-to-end tests. More on these outcomes is presented below.

9.4 Similarity Scoring and Schema Selection

The similarity scoring mechanism extends from the provision of the process to transform a database query from SQL into a multidimensional adjacency cube. From here, it is proposed to insert the process into the wider database query execution process, where upon receipt of a SQL query, this new process completes the transformation then compares, using the custom algorithm, the resulting cube with each previously-generated cube in a new query cache. This, combined with a weighting system, generates a range of reals, from 0 to 1, which are then placed along the number line in numerical order and the nearest K neighbours to the query at hand are selected (1) where K is an independent variable that can be continually adjusted according to the accuracy of the schema selection mechanism in a separate, asynchronous process.

Once K number of related queries have been isolated, a simple majority vote is used, reading the sub-schema assignment of each previous query and assigning the majority winner to the new query. An attempted execution of this query is tried against the selected sub-schema, supplying a query mapping sub-process to deal with syntactic difficulties, and the outcomes are measured as performance metrics. The query is then re-executed against the base schema asynchronously to establish baseline performance. These performance metrics are used to determine the degree of usefulness of the neighbouring K queries; if the performance is better using the base schema, a weighting system is used to reduce the weight of the queries; if the performance is better using the sub-schema, the weights of the neighbouring queries are increased. Due to the potential overhead sampling some percentage of queries is advocated for side-by-side performance comparison rather than running this process for each, and every, inbound query.

This process was implemented using a mixture of Python and SQL against PostgreSQL and Microsoft SQL Server, using a dataset detailing crime data in Chicago from which 4 sub-schemas of the base schema were constructed, constituting two partitions and two vertical shards.

The initial results were very promising. By connecting the query representation algorithm to the similarity scoring and schema mapping algorithm and running this against the Chicago data, it was found that for 5 sample queries, the expected vs. actual similarity scores differed by only 6.2%; in other words, the new algorithm classified these inbound queries nearly identically to human manual selection. The next stage was to test at scale.

The process was tested at scale in 10 test sets of 1,000 queries per test set, generated using the same methodology as used to test the query representation process and detailed in the previous section. Of these, it was found an estimated mean average failure ratio of 47:1000, leaving on average 953 valid queries per test run. Syntactic reliability was tested next; were the queries sound in the sense of having meaning within the schema, and returning rows? Here, difficulties were encountered; a failure ratio of 837:1000 was observed as some columns selected in mapped queries did not exist in the shard the process selected, and heavy system resource use was found (CPU, memory saturation) forcing the upgrade of the test system before proceeding. Several other issues were found which, together with the mapping error, were fixable by following an iterative fix, test, integrate strategy. After implementing the fixes, a failure rate of nil was achieved.

Next, two training sets of 1,000 queries each were generated, of which 954 and 955 queries were suitable for testing, respectively. A wide-ranging set of tests were defined, summarised in Chapter 7, running over 9,500 executions of the algorithm. Significant processing overhead was found to have contributed to system resource issues, also adding an average of 54ms to each query execution. However, discounting the processing overhead, a mean reduction in query execution time of 6.2% was achieved for all queries and for those queries run exclusively against sub-schemas, a reduction in execution time of 20.6%, a significant result.

Testing the K-adjustment process, it was found this was working as expected with the queries in the cache observed to have frequently-changing weights, the least-useful queries scoring progressively lower in the K-nearest neighbour and eventually aging out of the cache. It was discovered that weight adjustment process meant that the schema classifier was becoming progressively more accurate with a modest positive correlation across several thousand query samples. It was then possible to calculate the correlation formula use it to predict how quickly the system could be brought to a specified degree of accuracy in terms of the number of queries required to be presented.

The testing was not entirely positive. The query generator was limited in scope by the range of SQL queries that the new parsing mechanism can process; complex SQL, such as that containing CTEs, nested queries and side-effecting operators were out of scope, however JOINS are supported. A modest success of 20% query improvement was demonstrated, but some work is required on the implementation of the new mechanism since it was difficult to ascertain why more queries did not fall into scope of, and benefit from, the schema classifier. The new weighting mechanism is also

simple and could potentially be improved for more granularity, and the overall performance overhead of the algorithm requires that this implementation be refactored for better execution efficiency. Finally, it is noted that the implementation was put in place *alongside* an RDBMS, rather than *within* it; issues with proprietary formats and engine complexity mean the proper integration of this solution was not feasible for experimental purposes; in future work, it would be preferred that the code is refactored, significant efficiency improvements are made, the range of SQL in scope of processing is expanded (towards the full ANSI-SQL standard, if possible) and the system is fully integrated into the query engine to establish the full potential of this solution.

9.5 Dynamic Schema Redefinition

For the similarity scoring and schema selection process to function effectively, there must be a choice of schemas available. Typically, in a relational database platform, a single schema – a collection of tables – is presented, since they are physical representations of groups of data arranged in a sequential form. The sequential form may be contiguous storage, or it may be in a B*-tree arrangement, or some variant; but the table has permanence, and features such as two-stage locking, and the provision of different transaction isolation levels manage parallel access requests. Techniques such as partitioning and indexing coupled with broader strategies such as archival and infrastructural considerations help ensure access requests to these singular objects remain performant.

The proposed solution relies on the existence of multiple sub-schemas from the base schema; that is to say, derivations of the base schema, presented as separate logical objects. Using this description, this appears to be redefining the view, but the difference is that whereas views are logical representations of SQL queries which, when called upon, silently map the executing query to the base schema and return the results, the proposed solution is more akin to materialised views, which rely on the sub-schema definitions being separate database objects in their own right, with an independent existence, schema-bound to changes in the base schema. Yet, ideally, the solution would not mirror materialised views; it is proposed to use wholly logical definitions which, when called, do not map to the base schema but access the pages on which the data resides directly. This difference is subtle but important, since the execution plan used would reflect the sub-schema arrangement and therefore may perform better than the view, whose execution plan reflects the calls upon the underlying base schema objects. These sub-schemas may be supported with non-clustered indexes and other structures, but the data remains represented once and once only, and the base schema remains uninvolved.

The experimentation and testing undertaken fell short of this goal due to the lack of support for direct page lookups and accesses in any of the RDBMS platforms. Although these functions are integral parts of the database engine, they are not directly callable and therefore inaccessible unless the database engine source code, proprietary in many RDBMSs, can be accessed and amended. This remains a possibility for some open-source tooling such as PostgreSQL and MySQL. However, the closest analog was chosen to represent the direct access idea, the materialised view.

These algorithms were presented in Chapter 5 for the dynamic schema redefinition process, which consists of several components, shown in detail in Chapter 8. The query parser is responsible for accepting as input a database query in SQL form, and tokenising this query, identifying the attributes, predicates and relations within it. There is some overlap here with the query parser written for the query representation process since both components must shred a query to its component parts. Next, the information is written to temporary storage, and two processes come into play; the create and destroy MVs (materialised views, used in place of sub-schemas) phase assesses the contents of the temporary tables and based on execution count and other factors, creates appropriate sub-schemas that fit the queries. These are created in the target database. The plan is written to a new cache, and using similar logic to the feedback loop in the similarity scoring and schema mapping process, it is assessed whether the query would run faster against the sub-schema or the base schema. Performance data is collected on both cases and this is written to the cache. On re-presentation of the query to the engine, the query is checked for its presence in the cache and the preferred schema is noted. This preferred schema is used to run the query.

Interfacing with the previously-defined components, these schemas are under constant review by this process and destroyed when no longer used. The list of schemas available and the cache indicating chosen schema per query are combined with the cache from the similarity scorer and schema mapper. As the latter runs, weights for the various queries are adjusted. As the dynamic schema process runs, schemas, with an entirely abstract existence, are created and destroyed.

Throughout the testing, mixed results were found. It was found that the implementation of a working schema derivation algorithm was technically challenging; database queries, although using a finite syntax, have an extensive range of different forms and identification of sub-schemas was particularly error-prone. The TPC-C benchmark data set was used and against this schema, 9,660 distinct queries were generated. 1,462 queries were ran and cached before cache flush (a limit arrived at through trial-and-error) and of these, it was found 99.9% were suitable for mapping to new sub-schemas (MVs substituting here). From the queries, the process was able to group selections, predicates and relations and simplify and aggregate queries to create 53 new sub-schemas from which the testing was able to be based.

However, several practical issues arose. The MVs created only become MVs once indexes are created upon them; in Microsoft SQL Server, the test platform, a high failure rate was encountered

when implementing the MVs: only 6 of the 53 had indexes created successfully, with various limitations such as the use of non-key columns in OUTER JOINS prohibiting their use. This issue was overcome by creating fixed tables in lieu of MVs for these cases.

For the subset of queries suitable for the test harness, it was found that 19.3% of queries decreased in actual execution costs using this new technique. The implementation, like the implementation of the similarity calculator and schema selector, lacked full ANSI-SQL support which restricted the range of queries suitable for this approach to approximately one-tenth of all queries generated. It was also found that 4.7% of queries increased in query execution cost.

However, despite the limited range of queries, the indicative results were that this approach has some potential. The process to create and destroy new sub-schemas was demonstrated working (MV's and tables standing in) in real-time, a process not currently present in any RDBMS. Use of the new cache for schema mapping was demonstrated; the asynchronicity of the process was also demonstrated, allowing this to run alongside the ordinary cost-based query optimiser without interference in the core operations of the query engine; and the process of objectively assessing a given query for performance against multiple schemas was shown to be working. With refinement, it appears this process is viable for inclusion in RDBMS systems. With the same style of cache used in dynamic schema redefinition as the schema selector using the K-nearest neighbour process, it was also shown how the weight-based query classifier can work hand-in-hand with the schema redefinition process in schema mapping, although it is acknowledged that the testing did not include systemic end-to-end testing since the components were individually tested on different RDBMSs and using different test data sets.

The new efficiency measure was useful in assessing the access costs of queries for this component, and there is potential to develop this query measure as a universal measure, as query cost is currently heterogeneous and ill-defined across all RDBMS platforms.

More detail on the test outcomes was presented in Chapter 8.

9.7 Assessing the Research Questions, Aims and Objectives

The research questions, aims and objectives presented in Chapter 1, section 1.4 are now revisited to establish to what extent the stated goals have been met.

The original goals are presented in *italics* and commentary is presented underneath each item.

9.7.1 *Research questions*

- a. *As the demands of data processing have evolved from closed systems with known data structures driven by fixed schemas to open, unstructured systems driven by the applications, what disadvantages can be identified with the current object-relational database model given this evolution, and how can these be overcome?*

Through the literature review and qualitative research, a range of disadvantages were established with object-relational relationships, particularly the existence and taxonomy of object-relational impedance mismatch. This topic has wide coverage in the literature and both academic and industrial practitioners have commented extensively on the pitfalls in database performance that are manifest from this phenomenon. The literature review brought together the seminal sources and supplemented this with survey instruments, where a level of apathy and, to an extent, hostility was established to exist towards relational database platforms, partly as a result of the perceived difficulty of working with object-oriented sources. These negative artefacts were demonstrated, termed anti-patterns, through original primary research on two different RDBMSs, details of which were published in two separate conference proceedings.

Some existing mitigating tactics were established for overcoming these performance degradations, including the production of a list of recommendations for tuning ORM products; however, the original contribution in the form of PETAS is designed to, amongst other goals, reduce recompilations by normalising queries into adjacency matrices instead of using text-based parsers; encourage query re-use; and to present a method of using query subsets to improve performance against the backdrop of ORM-generated queries.

- b. *Can a new theory for query representation be developed as an alternative to representing queries as semantic objects? Is there an accompanying viable practical approach to implementing this new theory to overcome the disadvantages of storing and caching queries as non-comparable semantic objects and can this be used to improve the parsing and pre-optimisation stages of the query optimiser?*

The contribution of a new theory combined with the algorithms and sample implementation is one key component of the novel contribution to knowledge that this research produces. An entirely new technique was developed for query mapping, not previously represented in the literature, as a response to the research gap identified on the subject of improving query parsing efficiency. It was noted that current and historical parsers all, without exception, use text-based parsing methods, parse trees and object binding, neglecting the computational nature of SQL queries being extrusions of the relational algebra; an alternative approach was invented and demonstrated that added, on average, less than 2ms overhead to the query parser and resulted in a method of comparing queries that is superior for relative comparability than the current methods.

It was shown through experimentation that the outputs from the implemented process closely mirrored the similarity scores expected from a human expert; however it is acknowledged that there exist limitations in the range of ANSI-SQL that the parser is capable of handling, a limitation encountered for most of the qualitative research outputs. Expanding this range and refactoring the parser is a goal for future research and development of these ideas.

- c. *Can other approaches from alternative computational disciplines, such as machine learning, be applied to extend the current object-relational database storage and management methodologies, creating a responsive model that learns from system inputs to optimise system outputs?*

A new weighting system was created that was updateable in response to the relative execution times observed from running queries against a base schema and a sub-schema. This weighting mechanism was described in both theoretical and practical terms, providing the theory, the algorithms and the code listings, and experiments were designed and executed to validate whether the weights were updated as expected. It was found that the weighting mechanism worked as designed, with queries aging out for disuse as a result of plummeting weights correctly removed from the cache and queries with high comparative applicability to inbound queries having weights incremented as planned. This technique was K-nearest-neighbour with an updateable K-value, the process for which was also provided and tested successfully.

- d. *Can schema representation and usage in RDBMS systems be adjusted to incorporate more of the theoretical capabilities of axiomatic set theory, particularly the Zermelo-Fraenkel axiom of the schema of separation? Does such an approach work theoretically for query binding, and can such an approach be implemented in practice?*

The ZFC axiom schema of separation formalises the idea that separate subsets can form as derivations of base sets (or super/power-sets) and that these subsets are sets distinct from the parent set. Through the dynamic schema redefinition research component, separate subsets were defined distinct from the base sets; however, the implementation of the same suffered since the precise design that was aimed to implement – the logical representation of subsets accessing pages directly from disk, bypassing the base schema – was not feasible in current RDBMS systems since the source code for the query engine required extensive amendments. Therefore, only limited validation of this idea was performed, although the results were encouraging.

9.7.2. *Research aims*

- a. *To research the effects of object-relational impedance mismatch and associated factors, such as the impacts of big data that affect relational database query optimisation performance; to engage with the industry practitioner community to research the real-life performance consequences of queries generated from non-traditional sources, including ORM frameworks, upon relational databases.*

Through the literature review and qualitative research, the details of various anti-patterns emerging from ORMs were established and, using the survey outputs, commentary was offered upon the influence of the ‘Vs’ of big data upon the perceived performance of relational database platforms. The primary research outputs presented in Chapter 4 were used to demonstrate the ORM anti-patterns. In terms of qualitative research, this aim was not achieved in full. The selection of database practitioners was deficient in that the community of respondents did not include application developers who are most likely to use ORM products on a regular basis; fully half of the respondents did not have any meaningful insight or commentary to make upon ORM technology, and the interview participants were equally reticent on the subject. However, valuable opinions from the participants on performance issues encountered within database systems were extracted, and it was noted that non-traditional sources included data at volume (the first ‘V’ of big data) which, according to some respondents, their existing RDBMS platforms struggled to

cope with.

- c. *To identify and develop a novel solution to any adverse performance issues arising from these consequences; to test and validate the solution, and to establish an overarching design framework based upon this solution, detailed at both the theoretical and implementational level, to form the foundation of future work in developing the theoretical bases of this solution further.*

PETAS has been designed and developed; a multi-component model in response to performance difficulties arising from ORMs and big data challenges, which is detailed in this research. The overall success was mixed; most elements of the solution were demonstrated working, but further research is required for full implementation and integration into current RDBMSs. The tests conducted were extensive, employing primarily quantitative testing and the scientific, positivistic method; however, the qualitative research outputs were also valuable in validating and refining the problem definition and shaping the solution. Solution validation was only moderately successful; while there is confidence that the quantitative validation for each component and for some integrations (e.g., the query representation component, the similarity scoring component and the schema mapper component) were successful, end-to-end system testing was not fully completed due to a disparity in the platforms and the data sets used for the individual component testing.

Due to the difficulties of integrating this within an RDBMS engine, the original planned workshop/focus group approach for qualitative validation with database practitioners was not feasible; this was exacerbated by difficulties identifying a target set of participants for the same, given that object-relational impedance mismatch is a niche area of research. However, this research has been able to present a unified solution design under the PETAS umbrella and detail how the components integrate with quantitative experimental outputs bolstering the validation of the design.

9.7.3 Research objectives

- a. *To provide a summary review of the key technical underpinnings for the topics of this research, and to conduct a topical critical literature review of performance optimisation literature, both academic and professional, in the relational field together with related topics.*

This objective has been met by providing the review detailed in Chapters 2 and 3, forming a summary-based background and introductory review of the literature followed by a deep-divide topical and technical literature review in the second chapter.

- b. That the literature review in (a) encompasses the evolution of data in information systems; how data has been stored, categorised and measured, with emphasis on the trends and future developments required from data management frameworks to support these expectations.*

The literature review encompasses a historical overview of relational database systems and the contemporary view, noting new challenges. Topical sections were presented including the role of big data and changing landscapes on database concerns, and extensive research was carried out on the role of ORMs in database query performance.

- c. To investigate and identify weaknesses in current database design and query handling approaches, with particular emphasis on query representation and schema design.*

Through development of the query representation solution, a core weakness was identified; that queries are parsed as if they are textual objects, in the same way that natural language is parsed, without consideration that queries are extrusions of relational algebra using a finite syntax and as such are computable, comparable and could be represented in a form more suited to similarity analysis. This subject was explored in Chapters 5 and 6, culminating in a design centring on graph-theory inspired multidimensional adjacency matrices, for which the term ‘cubes’ was coined, and it was demonstrated how these query representations are comparable, whereas SQL query text is not.

- d. To validate any gaps identified in database performance optimisation research by collecting and analysing qualitative subjective data from industrial practitioners and from academic professionals.*

Two survey instruments were carried out; a questionnaire, targeted at database practitioners, which was formally structured using Likert scaling and free-form answers. From these results, the findings were arranged into themes and questions, and from this a second survey instrument was developed, the semi-structured interview, from which the questions derived from the questionnaire findings, triangulating the investigation. The interviews successfully uncovered opinions and experiences around working with data at

scale and responses were codified and classified using a grounded-theory-like approach, reaching a set of narrative conclusions detailed in Chapter 4.

- e. To identify suitable approaches to developing a conceptual solution to address the identified weaknesses, generalising this solution into a theoretical framework to augment current database storage designs, access methods, management processes and structural conventions, suitable for implementation across platforms.*

The PETAS framework is presented as a solution to identify the weaknesses in query representation, the weaknesses evidenced by ORMs in excessive recompilations and poor parsing and the weaknesses in relational systems in dealing with the variety of queries presented. This solution is presented theoretically in Chapter 5, and algorithmically and through experimental implementation in Chapters 6-9.

- f. To investigate if alternative computational optimisation tools and approaches, such as machine learning algorithms, can be used within a solution to the identified performance optimisation problems; if so, to present such a solution design and implementation.*

The proposed solution does not rely on machine learning techniques, and beyond the consideration of a machine-learning classifier, this research is not centrally concerned with the machinations of ML as a technique for augmenting database query performance. The discipline of ML was used as a toolbox of potential techniques, from which K-nearest neighbour was selected; to this extent, the objective has been met, but an extensive review of all possible techniques in the computer science domain was not undertaken to help achieve the goals; rather, following the pragmatic research philosophy discussed in Chapter 1, a ‘what-works’ approach was used and the solution was developed iteratively – a bottom-up, rather than top-down, spiral software development lifecycle.

- g. To evaluate the contributions of this research and propose new directions for further work based on the outcomes that were achieved.*

The conclusions are presented both on a chapter-by-chapter basis for each element of the primary and secondary research, and as a set of narrative conclusions in this chapter. In section 9.9 of this chapter, future research directions are discussed.

9.8 Future Research Directions

This research met many of the research questions, aims and objectives, however there exist various opportunities to correct, enhance and validate the proposed solution further.

First, further work is proposed to understand the extent of the ORM problem from the application development perspective. The omission of application developers as a potential target audience for the survey questions was an oversight which led to inferior data outcomes from the qualitative research in terms of ORM efficiencies; while the survey instruments had value, the literature review and other primary research were relied upon to qualify the ORM performance issues in more precise terms.

Secondly, further development of the query representation method and algorithmic implementation is recommended. The former should be developed further and proven against the relational algebra or the relational calculus; the latter should encompass the whole of the ANSI-SQL standard, in order to be viable and of practical use when integrating to mainstream RDBMS products. There are plenty of opportunities to develop the academic ideas underpinning this innovation, but also to, for example, develop and deploy this solution as an augmentation to an open-source RDBMS like PostgreSQL or MySQL as a product fork.

Thirdly, it is proposed to refactoring all the implementation code for the PETAS components for better intrinsic performance. The choice of Python was, it is believed, a contributor to substandard application performance, since lower-level languages have better memory and thread efficiency; both measures were observed consuming system resources in testing; it is also quite clear that these implementations could be implemented in more efficient code. Future work would include recreating these implementations with a wider range of tests to strengthen the validity of the findings.

Finally, in the broader sense, the research findings uncovered an appetite within the technical and business community for data storage and exploration tools capable of dealing with the demands of big data. Today's databases must be accommodating of unstructured or hybrid data types, be capable of withstanding high volumes of varying data presented at high velocity; must be performant, easy-to-use, understandable and accessible to a wide audience. On several of these points, current RDBMSs fail. Future work in the integration of those features inherent in non-relational databases to the relational database model, in the manner presented by original object-relational database proponents, may help to alleviate the apathy towards the relational database model that has driven many developers to using non-relational systems.

9.9 Chapter Summary

This chapter brought together the conclusions from the problem investigation, solution design and testing and validation. The components of PETAS were described in brief and the findings were discussed, with comments upon their validity and aptitude in meeting the stated goals. The research questions, aims and objectives were revisited and, for each item, commentary was offered on whether the item was met. The novel contributions to knowledge were considered and enumerated as outputs from the research. Future research directions were discussed and next steps outlined in improving the PETAS framework, drawing upon the lessons learned throughout this research project.

References follow. In the Appendices, the supplementary material is presented.

References

The reference format follows the Harvard (10th edition) ‘CiteThemRight’ variant supplied by Staffordshire University, available here: https://libguides.staffs.ac.uk/ld.php?content_id=9572296 (Accessed 26 February 2021).

Geographic locations of publishers are now omitted as per the latest Harvard guidance.

Chapter 1

- [1] SolidIT GmbH (2017). *DB-Engines Ranking - popularity ranking of database management systems*. Available at: <https://db-engines.com/en/ranking> (Accessed 26 June 2017).
- [2] Codd, E.F. (1970). ‘A relational model of data for large shared data banks’. *Communications of the ACM*, 13(6), pp.377-387.
- [3] Microsoft Corporation, u.d. *SQL Server technical documentation*. Available at: <https://docs.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver15> (Accessed 19 January 2021).
- [4] Oracle Corporation (2021). *Oracle Database*. Available at: <https://www.oracle.com/database/> (Accessed 19 Jan. 2021).
- [5] IBM Corporation (2021). *IBM DB2*. Available at: <https://www.ibm.com/uk-en/analytics/db2> (Accessed 19 Jan. 2021)
- [6] Ireland, C., Bowers, D., Newton, M. and Waugh, K. (2009). ‘A classification of object-relational impedance mismatch’. *First International Conference on Advances in Databases, Knowledge, and Data Applications*, pp. 36-43. Available at: <https://ieeexplore.ieee.org/abstract/document/5071809> (Accessed 18 November 2020).
- [7] Atzeni, P., Jensen, C.S., Orsi, G., Ram, S., Tanca, L. and Torlone, R. (2013). ‘The relational model is dead, SQL is dead, and I don't feel so good myself’. *ACM SIGMOD Record*, 42(2), pp.64-68.
- [8] Gigaom. 2017. *Facebook shares some secrets on making MySQL scale*. Available at: <https://gigaom.com/2011/12/06/facebook-shares-some-secrets-on-making-mysql-scale/> (Accessed 26 June 2017).
- [9] Sanders, G.L. and Shin, S. (2001). ‘Denormalization effects on performance of RDBMS’. *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, p. 3013-3018. Available at: <https://www.computer.org/csdl/proceedings/hicss/2001/0981/03/09813013.pdf> (Accessed 4 March 2019).
- [10] Karwin, B. (2017). *SQL Antipatterns*. Pragmatic Programmers.
- [11] Atzori, L., Iera, A. and Morabito, G. (2010). ‘The Internet of Things: a survey’. *Computer Networks*, 54(15), pp. 2787-2805.
- [12] Scuotto, V., Ferraris, A. and Bresciani, S. (2016). ‘Internet of Things’. *Business Process Management Journal*, 22(2), pp. 357-367.
- [13] Laplante, P.A. and Laplante, N. (2016). ‘The Internet of Things in Healthcare: Potential Applications and Challenges’. *IT Professional*, 18(3), pp. 2-4.
- [14] Yin, Y., Zeng, Y., Chen, X. and Fan, Y. (2016). ‘The internet of things in healthcare: an overview’. *Journal of Industrial Information Integration*, vol. 1, pp. 3-13.
- [15] Mabry, P.L. (2011). ‘Making Sense of the Data Explosion’. *American Journal of Preventive Medicine*, 40(5), pp. 159-161.
- [16] Degaut, M. (2016). ‘Spies and Policymakers: Intelligence in the Information Age’. *Intelligence and National Security*, 31(4), pp. 509-531.
- [17] Simmons, B.A. (2011). ‘International Studies in the Global Information Age’. *International Studies Quarterly*, 55(3), pp. 589-599.

- [18] Maule, A., Emmerich, W. and Rosenblum, D.S. (2008). 'Impact analysis of database schema changes'. *Proceedings of the 30th International Conference on Software Engineering*, pp. 451-460.
- [19] Søndergaard, J. (1970). 'Data Models and Query Languages'. *Communications of the ACM*, 13(6), pp.377-387.
- [20] Codd, E.F. (1971). 'A data base sublanguage founded on the relational calculus'. *Proceedings of the 1971 ACM SIGFIDET Workshop on Data Description, Access and Control*, pp. 35-68.
- [21] Beck, K., Grenning, J., Martin, R.C. et al. (2017). *Manifesto for Agile Software Development*. Available at: <http://agilemanifesto.org/> (Accessed 26 June 2017).
- [22] Ganesh Chandra, D. (2015). 'BASE analysis of NoSQL database'. *Future Generation Computer Systems*, vol. 52, pp. 13-21.
- [23] Atzeni, P., Bugiotti, F. & Rossi, L. (2014). 'Uniform access to NoSQL systems'. *Information Systems*, vol. 43, pp. 117-133.
- [24] Torres, A., Galante, R., Pimenta, M.S. and Martins, A.J.B., (2017). 'Twenty years of object-relational mapping: A survey on patterns, solutions and their implications on application design'. *Information and Software Technology*, vol. 82, pp. 1-18.
- [25] Poe, C. (2017). 'How the Database Will Hurt your Startup'. Available at: <https://www.linkedin.com/pulse/how-database-hurt-your-startup-curtis-poe> (Accessed 1 June 2017).
- [26] Creswell, J.W. (2003). *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. 2nd edn. Sage Publications.
- [27] Patton, M.Q. (1990). *Qualitative Evaluation and Research Methods*. 2nd edn. Sage Publications.
- [28] Ormerod, R. (2006). 'The History and Ideas of Pragmatism'. *Journal of the Operational Research Society*, 57(8), pp. 892-909.
- [29] Melles, G. (2008). 'New Pragmatism and the Vocabulary and Metaphors of Scholarly Design Research'. *Design Issues*, 24(4), pp. 88-101.
- [30] Schwandt, T.A. (1994). 'Constructivist, interpretivist approaches to Human Inquiry'. *Handbook of Qualitative Research*, vol. 1, pp.118-137.
- [31] Williams, M. (2000). 'Interpretivism and Generalisation'. *Sociology*, 34(2), pp.209-224.
- [32] Misak, C. (2013). 'Rorty, Pragmatism, and Analytic Philosophy'. *Humanities*, 2(3), pp. 369-383.
- [33] Beeri, C., Bernstein, P.A. and Goodman, N. (1978). 'A Sophisticate's Introduction to Database Normalization Theory'. *Proceedings of the 4th International Conference on Very Large Data Bases*, vol. 4, pp. 113-124.
- [34] Fagin, R. (1979). 'Normal Forms and Relational Database Operators'. *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pp. 153-160.
- [35] Lee, H. (1995). 'Justifying database normalization: a cost/benefit model'. *Information Processing and Management*, 31(1), pp.59-67.
- [36] Floridi, L. (ed). (2008). *The Blackwell Guide to the Philosophy of Computing and Information*. John Wiley & Sons.
- [37] Creswell, J. and Plano Clark, V. (2011). *Designing and Conducting Mixed Methods Research*. 2nd edn. Sage Publications.
- [38] Hesse-Biber, S.N. (2014). *Mixed Methods Research: Merging Theory with Practice*. The Guilford Press.
- [39] Saunders, M., Lewis, P. and Thornhill, A. (2009). *Research Methods for Business Students*. Pearson Education.
- [40] Lapan, S.D. (2012). *Qualitative Research: An Introduction to Methods and Designs*, pp. 41. Jossey-Bass.
- [41] Charmaz, K. (2014). *Constructing Grounded Theory*. Sage Publications, pp. 5-8.
- [42] Colley, D. and Stanier, C. (2017). 'Identifying New Directions in Database Performance Tuning'. *Procedia Computer Science*, vol. 121, pp. 260-265. Available at: <https://www.sciencedirect.com/science/article/pii/S1877050917322275> (Accessed 02 January 2021).
- [43] Date, C.J. (1990). *Relational Database Writings, 1985-1989, volume 1*. Addison Wesley.

- [44] Taylor, S.J., Bogdan, R. & Devault, M.L. (2016). *Introduction to Qualitative Research Methods: A Guidebook and Resource*. 4th edn. John Wiley & Sons.
- [45] Boehm, B.W. (1988). 'A Spiral Model of Software Development and Enhancement'. *Computer*, 21(5), pp.61-72.

Chapter 2

- [1] Whitmore, A., Agarwal, A. and Da Xu, L. (2015). 'The Internet of Things—A Survey of Topics and Trends'. *Information Systems Frontiers*, 17(2), pp.261-274. Available at: <https://link.springer.com/article/10.1007%2Fs10796-014-9489-2> (Accessed 4 March 2019).
- [2] Sharma, A., Schuhknecht, F.M. and Dittrich, J. (2018). 'The case for automatic database administration using deep reinforcement learning'. Preprint. *arXiv*. Available at: <https://arxiv.org/pdf/1801.05643.pdf> (Accessed 4 March 2019).
- [3] Abadi, D., Agrawal, R., Ailamaki, A., Balazinska, M. et al. (2014). 'The Beckman Report on Database Research'. *ACM SIGMOD Record*, 43(3), pp.61-70. Available at: <https://storage.googleapis.com/pub-tools-public-publication-data/pdf/44906.pdf> (Accessed 4 March 2019).
- [4] Elmasri, R. and Navathe, S. (2010). *Fundamentals of Database Systems*. Addison-Wesley Publishing Company.
- [5] Stonebraker, M. and Kemnitz, G. (1991). 'The POSTGRES Next-Generation Database Management System'. *Communications of the ACM*, 34(10), pp. 78-92. Available at: <https://doi.org/10.1145/125223.125262> (Accessed 19 May 2018).
- [6] Codd, E. (1970). 'A relational model of data for large shared data banks'. *Communications of the ACM*, 13 (6), pp. 377-387. Available at: <https://doi.org/10.1145/362384.362685> (Accessed 10 October 2016).
- [7] Stoll, R. (1979). *Set Theory and Logic*. Dover Publications.
- [8] Date, C. and Darwen, H. (2000). *Foundation for Future Database Systems: The Third Manifesto*. 2nd edn. Addison-Wesley.
- [9] Biskup, J., Dayal, U. and Bernstein, P.A. (1979). 'Synthesizing independent database schemas'. *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pp. 143-151. Available at: <https://doi.org/10.1145/582095.582118> (Accessed 06 June 2018).
- [10] Coronel, C. and Morris, S. (2016). *Database Systems: Design, Implementation and Management*. Cengage Learning.
- [11] Nayak, A., Poriya, A. and Poojary, D. (2013). 'Types of NOSQL databases and its comparison with relational databases'. *International Journal of Applied Information Systems*, 5(4), pp.16-19. Available at: https://www.researchgate.net/profile/Dikshay_Poojary/publication/302557703_Article_Type_of_nosql_databases_and_its_comparison_with_relational_databases/links/5aeaa2b50f7e9b837d3c40e7/Article-Type-of-nosql-databases-and-its-comparison-with-relational-databases.pdf (Accessed 21 April 2019).
- [12] International Standards Organisation (2016). *ISO/IEC 9075:2016*. Available at: <https://www.iso.org/standard/63555.html> (Accessed 27 February 2016).
- [13] Liu, Z.H. and Gawlick, D. (2015). 'Management of Flexible Schema Data in RDBMSs - Opportunities and Limitations for NoSQL'. *CIDR*. Available at: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper5.pdf (Accessed 4 March 2019).
- [14] Batra, R. (2018). 'A History of SQL and Relational Databases' in *SQL Primer*. Apress, pp. 183-187.
- [15] Bertino, E. and Sandhu, R. (2005). 'Database security - concepts, approaches, and challenges'. *IEEE Transactions on Dependable and Secure Computing*, 2(1), pp.2-19 . Available at: <https://search.proquest.com/openview/e03943036a308be889aa30aece031fe3/1?pq-origsite=gscholar&cbl=27603> (Accessed 24 October 2017).

- [16] Olivier, M.S. (2002). ‘Database privacy: balancing confidentiality, integrity and availability’. *ACM SIGKDD Explorations Newsletter*, 4(2), pp.20-27. Available at: <http://mo.co.za/open/dbpriv.pdf> (Accessed 4 March 2019).
- [17] Bjeladinovic, S. (2018). ‘A fresh approach for hybrid SQL/NoSQL database design based on data structuredness’. *Enterprise Information Systems*, 12(8-9), pp.1202-1220. Available at: <https://www.tandfonline.com/doi/abs/10.1080/17517575.2018.1446102> (Accessed 4 March 2019).
- [18] Liu, Z.H., Hammerschmidt, B., McMahon, D., Liu, Y. and Chang, H.J. (2016). ‘Closing the functional and performance gap between SQL and NoSQL’. *Proceedings of the 2016 International Conference on Management of Data*, pp. 227-238.
- [19] Agrawal, D., Chawla, S., Contreras-Rojas, B., Elmagarmid, A. et al. (2018). RHEEM: Enabling Cross-Platform Data Processing: May the Big Data be With You!. *Proceedings of the VLDB Endowment*, 11(11), pp.1414-1427. Available at: <https://www.ifi.uzh.ch/dam/jcr:002a7c16-46b4-497b-876f-ee5640c60a49/RHEEM.pdf> (Accessed 20 April 2019).
- [20] Ireland, C., Bowers, D., Newton, M. and Waugh, K. (2009). ‘A classification of object-relational impedance mismatch’. *First International Conference on Advances in Databases, Knowledge, and Data Applications*, pp. 36-43. Available at: <https://ieeexplore.ieee.org/abstract/document/5071809> (Accessed 18 November 2020).
- [21] Khan, M, Uddin, M. and Gupta N. (2014). ‘Seven V’s of Big Data; Understanding Big Data to extract value’. *Proceedings of the 2014 Zone 1 Conference of the American Society for Engineering Education: “Engineering Education: Industry Involvement and Interdisciplinary Trends”*. Available at: <http://doi.org/10.1109/ASEEZone1.2014.6820689> (Accessed 4 March 2019).
- [22] Molková, L. (2012). *Theory and Practice of Relational Algebra: Transforming Relational Algebra to SQL*. Lambert Academic Publishing.
- [23] Astrahan, M.M., Blasgen, M.W., Chamberlin, D.D., Eswaran, K.P., Gray, J.N., Griffiths, P.P., King, W.F., Lorie, R.A., McJones, P.R., Mehl, J.W. & Putzolu, G.R. (1976). ‘System R: Relational approach to database management’. *ACM Transactions on Database Systems*, 1(2), pp.97-137.
- [24] Ceri, S. and Gottlob, G. (1985). ‘Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries’. *IEEE Transactions on Software Engineering*, vol. SE-11(4), pp.324-345. Available at: <https://ieeexplore.ieee.org/abstract/document/1702016> (Accessed 13 April 2019).
- [25] Date, C.J. (1990). *Relational Database Writings, 1985-1989, volume 1*. Addison-Wesley.
- [26] Oracle Corporation (2019). *Oracle Database Online Documentation, 10g Release 2 (10.2)*. Available at: https://docs.oracle.com/cd/B19306_01/server.102/b14237/initparams035.htm#REFRN10025 (Accessed 27 February 2019).
- [27] Michel, D. and Microsoft Corporation (2019). *What is Parameter Sniffing?*. Available at: <https://blogs.technet.microsoft.com/mdegre/2011/11/06/what-is-parameter-sniffing/> (Accessed 27 February 2019).
- [28] Colley, D., Stanier, C., and Asaduzzaman, M. (2020). ‘Investigating the Effects of Object-Relational Impedance Mismatch on the Efficiency of Object-Relational Mapping Frameworks’. *Journal of Database Management*, 31(4). Available at: <https://www.igi-global.com/article/investigating-the-effects-of-object-relational-impedance-mismatch-on-the-efficiency-of-object-relational-mapping-frameworks/266402> (Accessed 18 January 2021)
- [29] Colley, D., Stanier, C. and Asaduzzaman, M. (2018). ‘The Impact of Object-Relational Mapping Frameworks on Relational Query Performance’. *International Conference on Computing, Electronics & Communications Engineering 2018 (ICCECE '18)*. Available at: <https://ieeexplore.ieee.org/document/8659222> (Accessed 18 January 2021).
- [30] Karwin, B. (2017). *SQL Antipatterns*. Pragmatic Bookshelf.
- [31] Meijer, E., Beckman, B. and Bierman, G. (2006). ‘LINQ: reconciling object, relations and XML in the .NET framework’. *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pp. 706-706.

- [32] Packer, A.N. (2001). *Configuring and Tuning Databases on the Solaris Platform*. Prentice Hall PTR.
- [33] Microsoft Corporation (2019). *Specify Query Parameterization Behavior by Using Plan Guides*. Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/performance/specify-query-parameterization-behavior-by-using-plan-guides?view=sql-server-2017> (Accessed 27 February 2019).
- [34] Trim, C. (2013). 'The Art of Tokenization'. *IBM Community (Language Processing)*. Available at: <https://www.ibm.com/developerworks/community/blogs/nlp/entry/tokenization?lang=en> (Accessed 20 February 2019).
- [35] Knuth, D. (1965). 'On the translation of languages from left to right'. *Information and Control*, vol. 8, pp 607-639. Available at: <https://www.sciencedirect.com/science/article/pii/S0019995865904262> (Accessed 06 February 2019).
- [36] Fritchey, G. (2018). *SQL Server Execution Plans*. 3rd edn. Redgate Software Limited.
- [37] Chaudhuri, S., Christensen, E., Graefe, G., Narasayya, V.R. and Zwillig, M.J. (1999). 'Self-tuning technology in Microsoft SQL Server'. *IEEE Data Engineering Bulletin*, 22(2), pp.20-26. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.654&rep=rep1&type=pdf#page=22> (Accessed 27 February 2019).
- [38] Freedman, C. (2006). *Scans vs. Seeks*. Available at: <https://blogs.msdn.microsoft.com/craigfr/2006/06/26/scans-vs-seeks/> (Accessed 13 April 2019).
- [39] Yu, C.T. and Meng, W. (1998). *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann.
- [40] Delaney, K. (2012). *Microsoft SQL Server Internals*. O'Reilly, ch. 12.
- [41] Microsoft Corporation (2015). *SQL Server: Max Degree of Parallelism (MDOP) and Affinity Mask*. Available at <https://social.technet.microsoft.com/wiki/contents/articles/25550.sql-server-max-degree-of-parallelism-mdop-and-affinity-mask.aspx> (Accessed 26 February 2019).
- [42] Haerder, T. and Reuter, A. (1983). 'Principles of transaction-oriented database recovery'. *ACM Computing Surveys (CSUR)*, 15(4), pp.287-317. Available at: <https://www.cs.utexas.edu/users/dsb/cs386d/Readings/Recovery/Principles-of-Recovery.pdf> (Accessed 26 February 2019).
- [43] Hameurlain, A. (2009). 'Evolution of query optimization methods: from centralized database systems to data grid systems'. *International Conference on Database and Expert Systems Applications*, pp. 460-470. Springer.
- [44] Waas, F. and Galindo-Legaria, C. (2000). 'Counting, enumerating, and sampling of execution plans in a cost-based query optimizer'. *ACM SIGMOD Record*, 29(2), pp. 499-509. Available at: [https://people.inf.elte.hu/kiss/cikkek/039%20Sampling%20of%20execution%20plans%20\(11%20oldal\).pdf](https://people.inf.elte.hu/kiss/cikkek/039%20Sampling%20of%20execution%20plans%20(11%20oldal).pdf) (Accessed 27 February 2019).
- [45] Kim, H., Ko, E., Young-Ho, J. and Lee, K. (2018). 'Migration from RDBMS to Column-Oriented NoSQL: Lessons Learned and Open Problems'. *Proceedings of the 7th International Conference on Emerging Databases*, pp. 25-33. Available at: https://doi.org/10.1007/978-981-10-6520-0_3 (Accessed 07 July 2019).
- [46] Moden, J. (2007). *Hidden RBAR: Triangular Joins*. Available at: <http://www.sqlservercentral.com/articles/T-SQL/61539> (Accessed 18 August 2018).
- [47] Chaudhuri, S. (1998). 'An overview of query optimization in relational systems'. *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 34-43. Available at: <https://doi.org/10.1145/275487.275492> (Accessed 19 March 2019).
- [48] Schiefer, K.B. and Valentin, G. (1999). 'DB2 universal database performance tuning'. *IEEE Data Engineering Bulletin*, 22(2), pp.12-19. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.30.7922&rep=rep1&type=pdf#page=14> (Accessed 27 February 2019).

- [49] Dageville, B., Das, D., Dias, K., Yagoub, K., Zait, M. and Ziauddin, M. (2004). ‘Automatic SQL tuning in Oracle 10g’. *Proceedings of the 30th International Conference on Very Large Data Bases*, vol. 30, pp. 1098-1109. Available at: <http://www.vldb.org/conf/2004/IND4P2.pdf> (Accessed 27 February 2019).
- [50] Pinto, Y. (2009). ‘A framework for systematic database de-normalization’. *Global Journal of Computer Science and Technology, Goa University*, pp.44-52. Available at: http://irgu.unigoa.ac.in/drs/bitstream/handle/unigoa/2295/Global_J_Comput_Sci_Tech_nol_9_44.pdf?sequence=1 (Accessed 4 March 2019).
- [51] Lee, H. (1995). ‘Justifying database normalization: a cost/benefit model’. *Information Processing and Management*, 31(1), pp.59-67. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.5174&rep=rep1&type=pdf> (Accessed 4 March 2019).
- [52] Gupta, A. and Mumick, I.S. (1995). ‘Maintenance of materialized views: Problems, techniques, and applications’. *IEEE Data Engineering Bulletin*, 18(2), pp.3-18. Available at: <https://ieeexplore.ieee.org/document/380392> (Accessed 5 March 2019).
- [53] Dragoni, N., Lanese, I., Larsen, S.T., Mazzara, M., Mustafin, R. and Safina, L. (2017). ‘Microservices: How to make your application scale’. *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pp. 95-104.
- [54] Thönes, J. (2015). Microservices. *IEEE Software*, 32(1), pp.116. Available at: <https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7030212> (Accessed 5 March 2019).
- [55] Chen, A. N. (1999). *Improving database performances in a changing environment with uncertain and dynamic information demand: An intelligent database system approach*. Doctoral dissertation. University of Connecticut. Available at: <https://opencommons.uconn.edu/dissertations/AAI9942566/> (Accessed 02 February 2017).
- [56] Terrizzano, I.G., Schwarz, P.M., Roth, M. and Colino, J.E. (2015). ‘Data Wrangling: The Challenging Journey from the Wild to the Lake’. *CIDR*. Available at: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper2.pdf (Accessed 4 March 2019).
- [57] Twitter Inc. (2014). *Manhattan, our real-time, multi-tenant distributed database for Twitter scale*. Available at: https://blog.twitter.com/engineering/en_us/a/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale.html (Accessed 6 March 2019).
- [58] Garcia-Molina, H., Ullman, J.D. and Widom, J. (2000). *Database System Implementation*. Prentice Hall. Ch. 5. Available at: <https://www.csd.uoc.gr/~hy460/pdf/000.pdf> (Accessed 02 February 2021).
- [59] Navathe, S., Ceri, S., Wiederhold, G. and Dou, J. (1984). ‘Vertical partitioning algorithms for database design’. *ACM Transactions on Database Systems (TODS)*, 9(4), pp.680-710. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.8306&rep=rep1&type=pdf> (Accessed 5 March 2019).
- [60] Microsoft Corporation (2018). *Understanding Isolation Levels*. Available at: <https://docs.microsoft.com/en-us/sql/connect/jdbc/understanding-isolation-levels?view=sql-server-2017> (Accessed 5 March 2019).
- [61] Sanders, G.L. and Shin, S. (2001). ‘Denormalization effects on performance of RDBMS’. *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, p. 3013-3018. Available at: <https://www.computer.org/csdl/proceedings/hicss/2001/0981/03/09813013.pdf> (Accessed 4 March 2019).
- [62] Hahnke, J. (1996). ‘Analysis applications add value: Three primary classes of analysis tools are: query and reporting, OLAP and data warehouse’. *Application Development Trends*, vol. 3, pp.38-46.
- [63] Kimball, R. and Ross, M. (2013). *The Data Warehouse Toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons.
- [64] Inmon, W.H., Imhoff, C. and Battas, G. (1999). *Building the Operational Data Store, volume 8*. John Wiley.

- [65] Batini, C., Lenzerini, M. and Navathe, S.B. (1986). ‘A comparative analysis of methodologies for database schema integration’. *ACM Computing Surveys (CSUR)*, 18(4), pp.323-364.

Chapter 3

- [1] Glaser, B.G. (1998). *Doing Grounded Theory: Issues and Discussions*. Sociology Press.
- [2] Shasha, D. (1996). ‘Tuning databases for high performance’. *ACM Computing Surveys (CSUR)*, 28(1), pp.113-115. Available at: <https://dl-acm-org.ezproxy.staffs.ac.uk/citation.cfm?id=234363> (Accessed 12 March 2019).
- [3] Microsoft Corporation (2018). *Best Practices for SQL Server in a SharePoint Server Farm*. Available at: <https://docs.microsoft.com/en-us/sharepoint/administration/best-practices-for-sql-server-in-a-sharepoint-server-farm> (Accessed 14 March 2019).
- [4] Ceri, S., Negri, M. and Pelagatti, G. (1982). ‘Horizontal data partitioning in database design’. *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, pp. 128-136.
- [5] Antova, L., Jansen, T., Koch, C. and Olteanu, D. (2008). ‘Fast and simple relational processing of uncertain data’. *IEEE 24th International Conference on Data Engineering*, pp. 983-992. Available at: <https://arxiv.org/pdf/0707.1644> (Accessed 13 March 2019).
- [6] Cornell, D.W. and Yu, P.S. (1990). ‘An effective approach to vertical partitioning for physical design of relational databases’. *IEEE Transactions on Software Engineering*, 16(2), pp.248-258. Available at: <https://ieeexplore.ieee.org/abstract/document/44388> (Accessed 13 March 2019).
- [7] Navathe, S., Ceri, S., Wiederhold, G. and Dou, J. (1984). ‘Vertical partitioning algorithms for database design’. *ACM Transactions on Database Systems (TODS)*, 9(4), pp.680-710.
- [8] Rodriguez, L. and Li, X. (2011). ‘A dynamic vertical partitioning approach for distributed database system’. 2011 IEEE International Conference on Systems Management and Cybernetics’, pp. 1853-1858. Available at: <https://ieeexplore.ieee.org/abstract/document/6083941> (Accessed 12 March 2019).
- [9] Microsoft Corporation, 2018. *SET TRANSACTION ISOLATION LEVEL (Transact-SQL)*. Available at: <https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-2017> (Accessed 13 March 2019).
- [10] Nehme, R. and Bruno, N. (2011). ‘Automated partitioning design in parallel database systems’. *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pp. 1137-1148. Available at: <https://cs.brown.edu/courses/cs227/archives/2012/papers/partitioning/p1137-nehme.pdf> (Accessed 14 March 2019).
- [11] Sundar, N. (2018). *Implementation of Sliding Window Partitioning in SQL Server to Purge Data*. Available at: <https://www.mssqltips.com/sqlservertip/5296/implementation-of-sliding-window-partitioning-in-sql-server-to-purge-data/> (Accessed 14 March 2019).
- [12] Martyn, T. (2004). ‘Reconsidering multi-dimensional schemas’. *ACM Sigmod Record*, 33(1), pp.83-88. . Available at: http://sigmodrecord.org/publications/sigmodRecord/0403/B6.Martyn_6page.pdf (Accessed 14 March 2019).
- [13] Lee, H. (1995). ‘Justifying database normalization: a cost/benefit model’. *Information Processing and Management*, 31(1), pp.59-67. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.5174&rep=rep1&type=pdf> (Accessed 14 March 2019).
- [14] Pinto, Y. (2009). ‘A framework for systematic database de-normalization’. *Global Journal of Computer Science and Technology*, pp. 44-53. Available at: http://irgu.unigoa.ac.in/drs/bitstream/handle/unigoa/2295/Global_J_Comput_Sci_Tech_nol_9_44.pdf?sequence=1 (Accessed 14 March 2019).
- [15] Sanders, G.L. and Shin, S. (2001). ‘Denormalization effects on performance of RDBMS’. *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, p.

- 3013-3018. Available at:
<https://www.computer.org/csdl/proceedings/hicss/2001/0981/03/09813013.pdf> (Accessed 4 March 2019).
- [16] Al-Barak, M. and Bahsoon, R. (2016). ‘Database design debts through examining schema evolution’. *IEEE 8th International Workshop on Managing Technical Debt (MTD)*, pp. 17-23. Available at: <https://ieeexplore.ieee.org/abstract/document/7776448> (Accessed 14 March 2019).
- [17] Date, C. (2012). *Database Design and Relational Theory: Normal Forms and All That Jazz*. O'Reilly.
- [18] International Standards Organisation (2016). *ISO/IEC 9075-1:2016 - Information Technology - Database languages - SQL - Part 1: Framework (SQL/Framework)*. Available at: <https://www.iso.org/standard/63555.html> (Accessed 14 August 2019).
- [19] Karwin, B. (2017). *SQL Antipatterns*. Pragmatic Bookshelf.
- [20] Kuznetsov, A. (2010). ‘Speeding up your queries with index covering’, in Nielson, P. et al. (eds), *SQL Server MVP Deep Dives, volume 1*. Manning Publications.
- [21] Zaniolo, C. (1984). ‘Database relations with null values’. *Journal of Computer and System Sciences*, 28(1), pp.142-166. Available at:
https://www.sciencedirect.com/science/article/pii/0022000084900801/pdf?md5=d16ed61057e90ca0177ace7bbb61fd6&pid=1-s2.0-0022000084900801-main.pdf&_valck=1 (Accessed 14 March 2019).
- [22] Bayer, R. (1972). ‘Symmetric binary B-trees: Data structure and maintenance algorithms’. *Acta Informatica*, 1(4), pp.290-306. Available at:
<https://link.springer.com/article/10.1007/BF00289509> (Accessed 10 January 2019).
- [23] Tripp, K. (2007). *The clustered index debate continues..* Available at:
<https://www.sqlskills.com/blogs/kimberly/the-clustered-index-debate-continues/> (Accessed 14 March 2019).
- [24] Graefe, G. (2011). ‘Modern B-tree techniques’. *Foundations and Trends® in Databases*, 3(4), pp.203-402. Available at:
<https://www.nowpublishers.com/article/DownloadSummary/DBS-028> (Accessed 07 July 2019).
- [25] Ozar, B. (2018). *Index Tuning Week: How Many Indexes Are Too Many?*. Available at:
<https://www.brentozar.com/archive/2018/10/index-tuning-week-how-many-indexes-are-too-many/> (Accessed 14 March 2019).
- [26] Catterall, R. (2010). *DB2 Indexes and Query Maintenance, Part 1*. Available at:
<https://www.ibmdatahub.com/blog/db2-indexes-and-query-performance-part-1> (Accessed 20 August 2019).
- [27] Lu, H., Ng, Y.Y. and Tian, Z. (2000). ‘T-tree or B-tree: Main memory database index structure revisited’. *Proceedings of the 11th Australasian Database Conference (ADC 2000)*, pp. 65-73.
- [28] Cooper, B.F., Sample, N., Franklin, M.J., Hjaltason, G.R. and Shadmon, M. (2001). ‘A fast index for semistructured data’. *Proceedings of the 27th VLDB Conference*, vol. 1, pp. 341-350). Available at: <http://www.vldb.org/conf/2001/P341.pdf> (Accessed 11 March 2018).
- [29] Guttman, A. (1984). ‘R-trees: a dynamic index structure for spatial searching’, 14(2), pp. 47-57. Available at: <https://apps.dtic.mil/dtic/tr/fulltext/u2/a168531.pdf> (Accessed 18 February 2019).
- [30] Trellis, T. (1987). ‘Index structures in computer programming’. John Wiley & Sons.
- [31] Fuhry, B., Bahmani, R., Brasser, F., Hahn, F., Kerschbaum, F. and Sadeghi, A.R. (2017). ‘HardIDX: Practical and secure index with SGX’. *IFIP Annual Conference on Data and Applications Security and Privacy*, pp. 386-408. Available at:
https://link.springer.com/chapter/10.1007/978-3-319-61176-1_22 (Accessed 28 December 2017).
- [32] Dziejczak, A., Wang, J., Das, S., Ding, B., Narasayya, V.R. and Syamala, M. (2018). ‘Columnstore and B+ tree-Are Hybrid Physical Designs Important?’. *Proceedings of the 2018 International Conference on Management of Data*, pp. 177-190. Available at:
<https://dl.acm.org/doi/abs/10.1145/3183713.3190660> (Accessed 20 December 2018).

- [33] Hanushevsky, A. and Nowak, M. (1999). ‘Pursuit of a scalable high performance multi-petabyte database’. *16th IEEE Symposium on Mass Storage Systems in cooperation with the 7th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp. 169-175. Available at: <https://www.computer.org/csdl/proceedings/mass/1999/0204/00/00830026.pdf> (Accessed 14 September 2019).
- [34] Morris, B. (2013). *A shared database is still an anti-pattern, no matter what the justification*. Available at: <https://www.ben-morris.com/a-shared-database-is-still-an-anti-pattern-no-matter-what-the-justification> (Accessed 30 November 2018).
- [35] Microsoft Corporation (2017). *Hints (Transact-SQL)*. Available at: <https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-table?view=sql-server-2017> (Accessed 13 March 2019).
- [36] Mohan, C., Pirahesh, H., Tang, W.G. and Wang, Y. (1994). ‘Parallelism in relational database management systems’. *IBM Systems Journal*, 33(2), pp.349-371. Available at: <https://ieeexplore.ieee.org/abstract/document/5387317> (Accessed 10 March 2019).
- [37] Microsoft Corporation (2018). *Columnstore indexes: Overview*. Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview?view=sql-server-2017> (Accessed 12 March 2019).
- [38] McGehee, B. (2011). *Do You Enable Lock Pages In Memory?*. Available at: <https://bradmcegee.com/2011/03/10/do-you-enable-lock-pages-in-memory/> (Accessed 08 April 2019).
- [39] Codd, E. (1970). ‘A relational model of data for large shared data banks’. *Communications of the ACM*, 13 (6), pp. 377-387. Available at: <https://doi.org/10.1145/362384.362685> (Accessed 10 October 2016).
- [40] Oracle Corporation (2019). *Database SQL Tuning Guide: SQL Processing*. Available at: https://docs.oracle.com/database/121/TGSQL/tgsql_sqlproc.htm#TGSQL175 (Accessed 06 February 2019).
- [41] Khurana, D., Koli, A., Khatter, K. and Singh, S. (2017). *Natural Language Processing: State of The Art, Current Trends and Challenges*. Preprint. *arXiv*. Available at: <https://arxiv.org/abs/1708.05148> (Accessed 10 December 2017).
- [42] Sun, S., Luo, C. and Chen, J. (2017). ‘A review of natural language processing techniques for opinion mining systems’. *Information Fusion*, vol. 36, pp.10-25.
- [43] Zelle, J.M. and Mooney, R.J. (1996). ‘Learning to parse database queries using inductive logic programming’. *Proceedings of the National Conference on Artificial Intelligence*, pp. 1050-1055. Available at: <http://www.aaai.org/Papers/AAAI/1996/AAAI96-156.pdf> (Accessed 15 February 2019).
- [44] Pardede, E., Rahayu, J.W. and Taniar, D. (2003). ‘New SQL standard for object-relational database applications’. *Proceedings of the 33rd European Solid-State Device Research - ESSDERC'03*, pp. 191-203.
- [45] Stonebraker, M., Brown, P. and Moore, D. (1999). *Object-Relational DBMSs: Tracking the Next Great Wave*. 2nd Edn. Morgan Kaufmann.
- [46] Feuerstein, S. and Pribyl, B. (1997). *Oracle PL/SQL Programming*. 2nd edn. O'Reilly.
- [47] Microsoft Corporation (2019). *Transact-SQL Reference (Database Engine)*. Available at: <https://docs.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-2017> (Accessed 06 February 2019).
- [48] Pitts, A. M. (2006). ‘Alpha-structural recursion and induction’. *Journal of the ACM*, 53(3), pp. 459-506.
- [49] Apache Software Foundation. (2019). *Apache Lucene*. Available at: <https://lucene.apache.org/> (Accessed 05 February 2019).
- [50] Nelson, P. and Search Technologies (2019). *Advanced Query Parsing Techniques*. [Online video]. Available at: <https://www.youtube.com/watch?v=abbBDLyFyS4> (Accessed 05 February 2019).
- [51] Covington, M.A. (2001). ‘A fundamental algorithm for dependency parsing’. *Proceedings of the 39th Annual ACM Southeast Conference*, pp. 95-102. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.7335&rep=rep1&type=pdf> (Accessed 05 February 2019).

- [52] Chomsky, N. (1956). ‘Three models for the description of language’. *IRE Transactions on Information Theory*, 2(3), pp.113-124. Archived version available at: <https://web.archive.org/web/20100919021754/http://chomsky.info/articles/195609--.pdf> (Accessed 06 February 2019).
- [53] Jackendoff, R. (1977). ‘X syntax: A study of phrase structure’. *Linguistic Inquiry Monographs*, vol. 2, pp.1-249.
- [54] Pachev, S. (2007). Understanding MySQL Internals. O'Reilly, ch. 9.
- [55] Oracle Corporation (2018). *MySQL-Server: sql/sql_lex.cc*. Online, GitHub repository. Available at: https://github.com/mysql/mysql-server/blob/8.0/sql/sql_lex.cc (Accessed 06 February 2019).
- [56] Estes, W. (2019). *Flex*. Online, GitHub repository. Available at: <https://github.com/westes/flex> (Accessed 02 December 2019).
- [57] Johnson, S.C. (1975). *Yacc: Yet another compiler-compiler*, vol. 32. Murray Hill and Bell Laboratories. Available at: <https://www.isi.edu/~pedro/Teaching/CSCI565-Fall15/Materials/Yacc.pdf> (Accessed 06 February 2019).
- [58] Free Software Foundation (2014). *GNU Operating System: GNU Bison*. Available at: <https://www.gnu.org/software/bison/> (Accessed 08 February 2019).
- [59] DeRemer, F.L. (1969). *Practical translators for LR (k) languages*. Doctoral dissertation. Massachusetts Institute of Technology. Available at: <https://core.ac.uk/download/pdf/81140495.pdf> (Accessed 06 February 2019).
- [60] Knuth, D. (1965). ‘On the translation of languages from left to right’. *Information and Control*, vol. 8, pp 607-639. Available at: <https://www.sciencedirect.com/science/article/pii/S0019995865904262> (Accessed 09 February 2019).
- [61] W3C (World Wide Web Consortium) (2013). *SPARQL 1.1 Overview*. Available at: <https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/> (Accessed 15 February 2019).
- [62] Jie, N. (2017). *Search engine traversal for data structures*. Available at: <https://www.cam.ac.uk/~jiejnd/search-engine-traversal-data-structures/> (Accessed 12 February 2019).
- [63] Mosharraf, M. and Taghiyareh, F. (2016). ‘Federated Search Engine for Open Educational Linked Data’. *Bulletin of the IEEE Technical Committee on Learning Technology*, 18(4), p.6. Available at: <http://tc.computer.org/tclt/wp-content/uploads/sites/5/2017/04/Mosharraf.pdf> (Accessed 15 February 2019).
- [64] Eldawy, A., Sabek, I., Elganainy, M., Bakeer, A., Abdelmotaleb, A., and Mokbel, M.F. (2017). ‘Sphinx: Empowering impala for efficient execution of SQL queries on big spatial data’. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10411 LNCS, pp. 65-83. Available at: https://link.springer.com/chapter/10.1007/978-3-319-64367-0_4 (Accessed 19 February 2019).
- [65] Ge, W., He, G. and Liu, X. (2018). ‘Business-oriented customized big data query system and its SQL parser design and implementation’. *MATEC Web of Conferences*, vol. 232, p.1004. Available at: https://www.matec-conferences.org/articles/mateconf/pdf/2018/91/mateconf_eitce2018_01004.pdf (Accessed 15 February 2019).
- [66] Li, C. and Gu, J. (2017). ‘A SQL transformation model of MongoDB based on ANTLR’. *Journal of Northwestern Polytechnical University*, 35(1), pp.143-147.
- [67] Cao, D. and Bai, D., (2010). ‘Design and implementation for SQL parser based on ANTLR’. *2nd International Conference on Computer Engineering and Technology (ICCET)*, vol. 4, pp. 274-276. Available at: <https://ieeexplore.ieee.org/abstract/document/5485593> (Accessed 15 February 2019).
- [68] Gudu Software (2019). *SQL Parse, Analyze, Transform and Format, All in One*. Available at: <http://www.sqlparser.com/index.php> (Accessed 15 February 2019).
- [69] Thenmozhi, D. and Aravindan, C. (2016). ‘Paraphrase identification by using clause-based similarity features and machine translation metrics’. *The Computer Journal*, 59(9),

- pp.1289-1302. Available at: <https://ieeexplore.ieee.org/abstract/document/8213052> (Accessed 14 February 2019).
- [70] Özsoyoğlu, G., Matos, V. and Özsoyoğlu, M. (1989). ‘Query processing techniques in the summary-table-by-example database query language’. *ACM Transactions on Database Systems (TODS)*, 14(4), pp.526-573.
- [71] Chamberlin, D.D., Astrahan, M.M., King, W.F., Lorie, R.A. et al. (1981). ‘Support for repetitive transactions and ad hoc queries in System R’. *ACM Transactions on Database Systems (TODS)*, 6(1), pp.70-94.
- [72] Microsoft Corporation (2019). *Optimize for Ad-Hoc Workloads Server Configuration Option*. Available at: <https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/optimize-for-ad-hoc-workloads-server-configuration-option?view=sql-server-2017> (Accessed 19 February 2019).
- [73] Li, J., Luong, M.-T., Jurafsky, D., and Hovy, E. (2015). ‘When are tree structures necessary for deep learning of representations?’. *Conference on Empirical Methods in Natural Language Processing*, pp. 2304-2314. Available at: <http://aclweb.org/anthology/D15-1278> (Accessed 15 February 2019).
- [74] Fagin, R., Kimelfield, B., Reiss, F. and Vansummeren, S. (2016). ‘Declarative Cleaning of Inconsistencies in Information Extraction’. *ACM Transactions on Database Systems (TODS)*, 41(1). Available at: <https://dl.acm.org/citation.cfm?id=2877202&dl=ACM&coll=DL> (Accessed 20 February 2019).
- [75] Trim, C. (2013). ‘The Art of Tokenization’. *IBM Community (Language Processing)*. Available at: <https://www.ibm.com/developerworks/community/blogs/nlp/entry/tokenization?lang=en> (Accessed 20 February 2019).
- [76] Webster, J.J. and Kit, C. (1992). ‘Tokenization as the initial phase in NLP’. 15th International Conference on Computational Linguistics, volume 4. Available at: <https://www.aclweb.org/anthology/C92-4173.pdf> (Accessed 10 May 2019).
- [77] Carpenter, B. (2005). *The Logic of Typed Feature Structures: With Applications to Unification Grammars, Logic Programs and Constraint Resolution*. Cambridge University Press.
- [78] Ireland, C., Bowers, D., Newton, M. and Waugh, K. (2009). ‘A classification of object-relational impedance mismatch’. *First International Conference on Advances in Databases, Knowledge, and Data Applications*, pp. 36-43. Available at: <https://ieeexplore.ieee.org/abstract/document/5071809> (Accessed 18 November 2020).
- [79] Phillippi, S. (2005). ‘Model-driven generation and testing of object-relational mappings’. *Journal of Systems and Software*, 77(2), pp. 193-207. Available at: <https://doi.org/10.1016/j.jss.2004.07.252> (Accessed 04 November 2018).
- [80] Chen, A. N. (1999). *Improving database performances in a changing environment with uncertain and dynamic information demand: An intelligent database system approach*. Doctoral dissertation. University of Connecticut. Available at: <https://opencommons.uconn.edu/dissertations/AAI9942566/> (Accessed 18 January 2020).
- [81] Colley, D., Stanier, C. and Asaduzzaman, M. (2018). ‘The Impact of Object-Relational Mapping Frameworks on Relational Query Performance’. *International Conference on Computing, Electronics & Communications Engineering 2018 (ICCECE '18)*. Available at: <https://ieeexplore.ieee.org/document/8659222> (Accessed 18 January 2021).
- [82] Colley, D. and Stanier, C. (2017). Identifying New Directions in Database Performance Tuning. *Procedia Computer Science*, vol. 121, pp.260-265. Available at: <https://www.sciencedirect.com/science/article/pii/S1877050917322275> (Accessed 18 January 2021).
- [83] Colley, D., Stanier, C., and Asaduzzaman, M. (2020). ‘Investigating the Effects of Object-Relational Impedance Mismatch on the Efficiency of Object-Relational Mapping Frameworks’. *Journal of Database Management*, 31(4). Available at: <https://www.igi-global.com/article/investigating-the-effects-of-object-relational-impedance-mismatch-on-the-efficiency-of-object-relational-mapping-frameworks/266402> (Accessed 18 January 2021).

- [84] Chen, T., Shang, W., Zhen, M.J., Hassan, A.E., Nasser, M. and Flora, P. (2014). ‘Detecting performance anti-patterns for applications developed using object-relational mapping’. *Proceedings of the 36th International Conference on Software Engineering*, pp. 1001-1012. Available at: <https://doi.org/10.1145/2568225.2568259> (Accessed 16 January 2020).
- [85] Microsoft Corporation, Narumoto, M., Bennage, C. and Wasson, M. (2017). *Extraneous Fetching Antipattern*. Available at: <https://docs.microsoft.com/en-us/azure/architecture/antipatterns/extraneousfetching> (Accessed 10 January 2020).
- [86] Dennis, A., Wixom, B.H. and Tegarden, D. (2015). *Systems analysis and design: An object-oriented approach with UML*. John Wiley & Sons.
- [87] Kay, A.C. (1996). ‘The early history of Smalltalk’. *History of Programming Languages*, vol. 2, pp. 511-598. Available at: <https://doi.org/10.1145/234286.1057828> (Accessed 20 February 2020).
- [88] Suppes, P. (1960). *Axiomatic set theory*. Courier Corporation.
- [89] Kim, H., Ko, E., Jeon, Y. and Lee, K. (2018). ‘Migration from RDBMS to Column-Oriented NoSQL: Lessons Learned and Open Problems’. *Proceedings of the 7th International Conference on Emerging Databases*, pp. 25-33. Available at: https://doi.org/10.1007/978-981-10-6520-0_3 (Accessed 19 February 2020).
- [90] Moden, J. (2007). *Hidden RBAR: Triangular joins*. Available at: <http://www.sqlservercentral.com/articles/T-SQL/61539> (Accessed 18 February 2021).
- [91] Cheung, A., Madden, S. & Solar-Lezama, A. (2016). ‘Sloth: Being lazy is a virtue (when issuing database queries)’. *ACM Transactions on Database Systems (TODS)*, 41(2), p.8.
- [92] Fritchey, G. (2018). *SQL Server 2017 Query Performance Tuning*. Apress, Ch. 7.
- [93] Date, C. J. (1990). *Relational Database Writings, 1985-1989, volume 1*. Addison-Wesley.
- [94] Microsoft Corporation (2009). *Getting Started with Entity Framework 6 Code First using MVC 5*. Available at: <https://docs.microsoft.com/en-us/aspnet/mvc/overview/gettingstarted/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application> (Accessed 10 January 2021).
- [95] Małysiak-Mrozek, B., Mazurkiewicz, H. and Mrozek, D. (2019). ‘Incorporating Fuzzy Logic in Object-Relational Mapping Layer for Flexible Medical Screenings’. *Intelligent Methods and Big Data in Industrial Applications*, pp. 213-233.
- [96] Raghu, R. and Varma, N.S. (2018). ‘JSON as ORM Mapping Database Layer for the SaaS-Based Multi-tenant Application’ *Recent Findings in Intelligent Computing Techniques* pp. 295-306. Available at: https://link.springer.com/chapter/10.1007/978-981-10-8633-5_30 (Accessed 15 March 2019).
- [97] Wikipedia, n.d. *List of ORM mapping frameworks*. Available at: https://en.wikipedia.org/wiki/List_of_object-relational_mapping_software (Accessed 15 March 2019).
- [98] Yannakakis, M. (1990). ‘Graph-theoretic methods in database theory’. *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 230-242. Available at: https://www.researchgate.net/profile/Mihalis_Yannakakis/publication/221559463_Graph-Theoretic_Methods_in_Database_Theory/links/57adf07e08ae15c76cb34ccf.pdf (Accessed 19 March 2019).
- [99] Mihalcea, R. and Radev, D. (2011). *Graph-based natural language processing and information retrieval*. Cambridge University Press.
- [100] Alqaryouti, O., Khwileh, H., Farouk, T. A., Nabhan, A. R. and Shaalan, K. (2018). *Graph-based keyword extraction*. Available at: https://www.researchgate.net/publication/321150259_Graph-Based_Keyword_Extraction (Accessed 20 March 2019).
- [101] Matsuo, Y. and Ishizuka, M. (2004). *International Journal on Artificial Intelligence Tools*, 13(01), pp.157-169. Available at: <https://www.aaai.org/Papers/FLAIRS/2003/Flairs03-076.pdf> (Accessed 25 February 2021).
- [102] Robinson, I., Webber, J. and Eifrem, E. (2013). *Graph databases*. 2nd edn. O'Reilly.

- [103] Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y. and Wilkins, D. (2010). ‘A comparison of a graph database and a relational database: a data provenance perspective’. *Proceedings of the 48th Annual Southeast Regional Conference*, p. 42. Available at: https://john.cs.olemiss.edu/~ychen/publications/conference/vicknair_acmse10.pdf (Accessed 19 March 2019).
- [104] Harary, F. (1962). ‘The determinant of the adjacency matrix of a graph’. *Siam Review*, 4(3), pp.202-210. Available at: <https://epubs.siam.org/doi/abs/10.1137/1004057> (Accessed 21 May 2020).
- [105] Gallo, G., Longo, G., Pallottino, S. and Nguyen, S. (1993). ‘Directed hypergraphs and applications’. *Discrete Applied Mathematics*, 42(2-3), pp.177-201.
- [106] Reutter, J.L., Romero, M. and Vardi, M.Y. (2017). ‘Regular queries on graph databases’. *Theory of Computing Systems*, 61(1), pp.31-83. Available at: <http://repositorio.uchile.cl/bitstream/handle/2250/148293/Regular-Queries-on-Graph-Databases.pdf?sequence=1> (Accessed 19 March 2019).
- [107] AlgoWiki (2018). *Transitive closure of a directed graph*. Available at: https://algowiki-project.org/en/Transitive_closure_of_a_directed_graph (Accessed 19 March 2019).
- [108] Daniel, G., Sunyé, G. and Cabot, J. (2016). ‘UML to Graph DB: mapping conceptual schemas to graph databases’. *International Conference on Conceptual Modeling*, pp. 430-444. Available at: <https://hal.archives-ouvertes.fr/hal-01344015/document> (Accessed 20 March 2019).
- [109] Zheng, W., Zou, L., Lian, X., Wang, D. and Zhao, D. (2015). ‘Efficient graph similarity search over large graph databases’. *IEEE Transactions on Knowledge and Data Engineering*, 27(4), pp.964-978. Available at: <https://faculty.utrgv.edu/xiang.lian/papers/TKDE15-weiguo.pdf> (Accessed 19 March 2019).
- [110] Nawaz, M., Khan, S., Qureshi, R. and Yan, H. (2019). ‘Clustering based one-to-one hypergraph matching with a large number of feature points’. *Signal Processing: Image Communication*. Available at: <https://www.sciencedirect.com/science/article/abs/pii/S0923596518306155> (Accessed 19 March 2019).
- [111] Emmert-Streib, F., Dehmer, M. and Shi, Y. (2016). ‘Fifty years of graph matching, network alignment and network comparison’. *Information Sciences*, vol. 346, pp.180-197.
- [112] Vento, M. (2015). ‘A long trip in the charming world of graphs for pattern recognition. *Pattern Recognition*’, 48(2), pp.291-301.
- [113] Perchant, A. and Bloch, I. (2002). ‘Fuzzy morphisms between graphs’. *Fuzzy Sets and Systems*, 128(2), pp.149-168. Available at: <https://perso.telecom-paristech.fr/bloch/papers/FSS-Aymeric.pdf> (Accessed 19 March 2019).

Chapter 4

- [1] Chen, A. N. (1999). *Improving database performances in a changing environment with uncertain and dynamic information demand: An intelligent database system approach*. Doctoral dissertation. University of Connecticut. Available at: <https://opencommons.uconn.edu/dissertations/AAI9942566/> (Accessed 20 February 2021).
- [2] Colley, D., Stanier, C. and Asaduzzaman, M. (2018). ‘The Impact of Object-Relational Mapping Frameworks on Relational Query Performance’. *International Conference on Computing, Electronics & Communications Engineering 2018 (ICCECE '18)*. Available at: <https://ieeexplore.ieee.org/document/8659222> (Accessed 18 Jan. 2021).
- [3] Colley, D., Stanier, C., and Asaduzzaman, M. (2020). ‘Investigating the Effects of Object-Relational Impedance Mismatch on the Efficiency of Object-Relational Mapping Frameworks’. *Journal of Database Management*, 31(4). Available at: <https://www.igi-global.com/article/investigating-the-effects-of-object-relational-impedance-mismatch-on-the-efficiency-of-object-relational-mapping-frameworks/266402> (Accessed 18 January 2021).

- [4] Clarke, V. and Braun, V. (2013). ‘Teaching thematic analysis: Overcoming challenges and developing strategies for effective learning’. *The Psychologist*, 26(2).
- [5] Aronson, J. (1994). ‘A pragmatic view of thematic analysis’. *The Qualitative Report*, 2(1). Archived version available at: <http://web.archive.org/web/20000303144031/http://www.nova.edu/ssss/QR/BackIssues/QR2-1/aronson.html> (Accessed 25 February 2021).
- [6] Ireland, C., Bowers, D., Newton, M. and Waugh, K. (2009). ‘A classification of object-relational impedance mismatch’. *First International Conference on Advances in Databases, Knowledge, and Data Applications*, pp. 36-43. Available at: <https://ieeexplore.ieee.org/abstract/document/5071809> (Accessed 18 November 2020).
- [7] Ambler, S. (2003). *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley Publishing.
- [8] Ambler, S., Sadalage, P.J. (2006). *Refactoring Databases: Evolutionary Database Design*. Addison Wesley Professional.
- [9] Niu, B., Martin, P., Powley, W., Horman, R. and Bird, P. (2006). ‘Workload adaptation in autonomic DBMSs’. *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 13-23. Available at: <https://dl.acm.org/doi/abs/10.1145/1188966.1188984> (Accessed 10 January 2020).
- [10] Vial, G. (2015). ‘Database refactoring: Lessons from the trenches’. *IEEE Software*, 32(6), pp.71-79. Available at: <https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7310988> (Accessed 15 February 2021).
- [11] otter.ai, 2020. Untitled. Available at: <https://www.otter.ai> (Accessed 16 November 2020).
- [12] QSR International (2020). NVivo Qualitative Data Analysis Software. Available at: <https://www.qsrinternational.com/nvivo-qualitative-data-analysis-software/home> (Accessed 23 February 2021).
- [13] Microsoft Corporation, u.d. ‘Getting Started with Entity Framework 6 Code First using MVC 5’. Available at: <https://docs.microsoft.com/en-us/aspnet/mvc/overview/gettingstarted/getting-started-with-ef-using-mvc/creating-an-entityframework-data-model-for-an-asp-net-mvc-application> (Accessed 09 May 2018).
- [14] Pacific Marine Environmental Laboratory (2018). *1997-98 El Nino Sea Level: Monthly Mean Sea Level for U.S. West Coast, Alaska, Hawaii*. Available at: https://nctr.pmel.noaa.gov/Sea_level_1997_98/sea_level_1997_98.html (Accessed 08 June 2017).
- [15] Galt, J.A., Overland, J.E., Pease, C.H. and Stewart, R.J. (1978). ‘Numerical Studies – Pacific Marine Environmental Laboratory’. *Outer Continental Shelf Environmental Assessment, Program Research Unit 40*. Available at: <https://espis.boem.gov/final%20reports/1479.pdf> (Accessed 10 June 2017).
- [16] Karwin, B. (2017). *SQL Antipatterns*. Pragmatic Bookshelf.
- [17] Fritchey, G. (2018). *SQL Server Execution Plans*. 3rd edn. Redgate Software Limited.
- [18] Wikipedia, u.d. *Spherical Trigonometry*. Available at: https://en.wikipedia.org/wiki/Spherical_trigonometry (Accessed 04 May 2020).
- [19] Alvarez, R. and Urla, J. (2002). ‘Tell me a good story: using narrative analysis to examine information requirements interviews during an ERP implementation’. *ACM SIGMIS Database: the DATABASE for Advances in Information Systems*, 33(1), pp.38-52.

Chapter 5

- [1] Dennis, A., Wixom, B.H. and Tegarden, D. (2015). *Systems analysis and design: An object-oriented approach with UML*. John Wiley & Sons.
- [2] Kay, A.C. (1996). ‘The early history of Smalltalk’. *History of Programming Languages*, vol. 2, pp. 511-598. Available at: <https://doi.org/10.1145/234286.1057828> (Accessed 20 February 2020).
- [3] Suppes, P. (1960). *Axiomatic set theory*. Courier Corporation.

- [4] Ireland, C., Bowers, D., Newton, M. and Waugh, K. (2009). ‘A classification of object-relational impedance mismatch’. *First International Conference on Advances in Databases, Knowledge, and Data Applications*, pp. 36-43. Available at: <https://ieeexplore.ieee.org/abstract/document/5071809> (Accessed 18 November 2020).
- [5] Phillippi, S. (2005). ‘Model-driven generation and testing of object-relational mappings’. *Journal of Systems and Software*, 77(2), pp. 193-207. Available at: <https://doi.org/10.1016/j.jss.2004.07.252> (Accessed 04 November 2018).
- [6] Chen, T., Shang, W., Zhen, M.J., Hassan, A.E., Nasser, M. and Flora, P. (2014). ‘Detecting performance anti-patterns for applications developed using object-relational mapping’. *Proceedings of the 36th International Conference on Software Engineering*, pp. 1001-1012. Available at: <https://doi.org/10.1145/2568225.2568259> (Accessed 16 January 2020).
- [7] Microsoft Corporation, Narumoto, M., Bennage, C. and Wasson, M. (2017). *Extraneous Fetching Antipattern*. Available at: <https://docs.microsoft.com/en-us/azure/architecture/antipatterns/extraneousfetching> (Accessed 10 January 2020).
- [8] Delaney, K. (2013). *Microsoft SQL Server 2012 Internals*. Microsoft Press.
- [9] Microsoft Corporation (2017). *Clustered and Nonclustered Indexes Described*. Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-2017> (Accessed 20 February 2021).
- [10] Oracle Corporation (2018). *Managing Indexes (Oracle Database Online Documentation 10g)*. Available at: https://docs.oracle.com/cd/B19306_01/server.102/b14231/indexes.htm (Accessed 20 February 2021).
- [11] Sellis, T.K. (1988). ‘Intelligent caching and indexing techniques for relational database systems’. *Information Systems*, 13(2), pp.175-185. Available at: <https://www.sciencedirect.com/science/article/abs/pii/0306437988900142> (Accessed 21 February 2021).
- [14] Chen, A. N. (1999). *Improving database performances in a changing environment with uncertain and dynamic information demand: An intelligent database system approach*. Doctoral dissertation. University of Connecticut. Available at: <https://opencommons.uconn.edu/dissertations/AAI9942566/> (Accessed 02 February 2017).
- [15] Batini, C., Lenzerini, M. and Navathe, S.B. (1986). ‘A comparative analysis of methodologies for database schema integration’. *ACM Computing Surveys (CSUR)*, 18(4), pp. 323-364. Available at: <https://doi.org/10.1145/27633.27634> (Accessed 20 February 2021).
- [16] Pinkel, C., Binnig, C., Jiménez-Ruiz, E., May, W. et al. (2015). ‘RODI: A benchmark for automatic mapping generation in relational-to-ontology data integration’. *European Semantic Web Conference (ESWC '15)*, pp. 21-37. Available at: https://doi.org/10.1007/978-3-319-18818-8_2 (Accessed 25 February 2021).
- [17] Dedić, N. and Stanier, C. (2016). ‘Towards differentiating business intelligence, big data, data analytics and knowledge discovery’. *International Conference on Enterprise Resource Planning Systems*, vol. 285, pp. 114-122. Available at: https://link.springer.com/chapter/10.1007/978-3-319-58801-8_10 (Accessed 25 February 2021).
- [18] Gandomi, A. and Haider, M. (2015). ‘Beyond the hype: Big data concepts, methods, and analytics’. *International Journal of Information Management*, 35(2), pp. 137-144. Available at: <https://doi.org/10.1016/j.ijinfomgt.2014.10.007> (Accessed 25 February 2021).
- [19] Hamming, R.W. (1950). ‘Error detecting and error correcting codes’. *Bell Labs Technical Journal*, 29(2), pp. 147-160. Available at: <https://doi.org/10.1002/j.1538-7305.1950.tb00463.x> (Accessed 26 February 2021).
- [20] Codd, E.F. (1970). ‘A relational model of data for large shared data banks’. *Communications of the ACM*, 13(6), pp.377-387.
- [21] Molková, L. (2012). *Theory and Practice of Relational Algebra: Transforming Relational Algebra to SQL*. Lambert Academic Publishing.
- [22] Chaudhuri, S. (1998). ‘An overview of query optimization in relational systems’. *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of*

- Database Systems*, pp. 34-43. Available at: <https://doi.org/10.1145/275487.275492> (Accessed 23 February 2021).
- [23] Pachev, S. (2007). *Understanding MySQL Internals*. O'Reilly, Ch. 9.
- [24] Nevarez, B. (2010). *Inside the SQL Query Optimizer*. Simple Talk Publishing.
- [25] Brualdi, R.A. and Ryser, H. (1991). *Combinatorial matrix theory*. Cambridge University Press.
- [26] Feynman, R., eds. Hey, A.J. and Allen, R.W. (1999). *Feynman: Lectures on Computation*. Penguin. Ch. 4, pp. 111-113.
- [27] Harary, F. (1962). 'The determinant of the adjacency matrix of a graph'. *Siam Review*, 4(3), pp. 202-210. *Society for Industrial and Applied Mathematics*. Available at: <https://doi.org/10.1137/1004057> (Accessed 23 February 2021).
- [28] Oracle Corporation (2018). *SQL Processing (Oracle Database Online Documentation 12g Release 1)* Available at: https://docs.oracle.com/database/121/TGSQL/tgsql_sqlproc.htm#TGSQL175 (Accessed 22 February 2021).
- [29] Mazumdar, P., Agarwal, S. and Banerjee, A. (2016). 'Azure SQL Database: Performance and Monitoring' in *Pro SQL Server on Microsoft Azure*. Apress.
- [30] Chang, J. (2019). *Execution Plan Cost Model*. Available at: <http://www.qdpma.com/CBO/PlanCost.html> (Accessed 02 October 2020).
- [31] Microsoft Corporation (2020). *SQL Server Index Architecture and Design Guide*. Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide?view=sql-server-ver15> (Accessed 28 May 2020).
- [32] Oracle Corporation (2015). *Indexes and Index-Organised Tables*. Available at: https://docs.oracle.com/cd/E11882_01/server.112/e40540/indexiot.htm#CNCPT721 (Accessed 28 May 2020).
- [33] Korotkevitch, D. (2014). *Pro SQL Server Internals*. Apress, pp. 125-148.
- [34] Strate, J. and Krueger, T. (2012). *Expert Performance Indexing for SQL Server 2012*. Apress, pp. 51-89.
- [35] Fritchey, G. and Dam, S. (2009). 'Execution Plan Cache Analysis' in *SQL Server 2008 Query Performance Tuning Distilled*. Springer, pp. 241- 281.

Chapter 6

- [1] Young, N. (1988). *An Introduction to Hilbert Space*. Cambridge University Press.
- [2] Knuth, D. (1965). 'On the translation of languages from left to right'. *Information and Control*, vol. 8, pp 607-639. Available at: <https://www.sciencedirect.com/science/article/pii/S0019995865904262> (Accessed 06 February 2019).
- [3] Free Software Foundation (2014). *GNU Operating System: GNU Bison*. Available at: <https://www.gnu.org/software/bison/> (Accessed 06 February 2019).

Chapter 7

- [1] City of Chicago (2017). *Public Safety dataset*. Available at: <https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-present/ijzp-q8t2> (Accessed 18 October 2017).

Chapter 8

- [1] Colley, D., Stanier, C., and Asaduzzaman, M. (2020). 'Investigating the Effects of Object-Relational Impedance Mismatch on the Efficiency of Object-Relational Mapping

- Frameworks’. *Journal of Database Management*, 31(4). Available at: <https://www.igi-global.com/article/investigating-the-effects-of-object-relational-impedance-mismatch-on-the-efficiency-of-object-relational-mapping-frameworks/266402> (Accessed 18 January 2021)
- [2] Suppes, P. (1960). *Axiomatic set theory*. Courier Corporation.
- [3] Yourdon, E. (1989). *Modern Structured Analysis*. Prentice-Hall.
- [4] Everest, G.C. (1976). ‘Basic data structure models explained with a common example’. *Proceedings of the Fifth Texas Conference on Computing Systems*, pp. 18-19. Available at: https://www.researchgate.net/profile/Gordon-Everest-2/publication/291448084_BASIC_DATA_STRUCTURE_MODELS_EXPLAINED_WITH_A_COMMON_EXAMPLE/links/57affb4b08ae95f9d8f1ddc4/BASIC-DATA-STRUCTURE-MODELS-EXPLAINED-WITH-A-COMMON-EXAMPLE.pdf (Accessed 23 February 2021).
- [5] Microsoft Corporation (2017). *SQL Server Plan Cache Object*. Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/performance-monitor/sql-server-plan-cache-object?view=sql-server-ver15> (Accessed 29 January 2021).
- [6] TPC, 2010. *TPC-C*. Available at: <http://www.tpc.org/tpcc/> (Accessed 07 February 2020).
- [7] HammerDB, 2020. Untitled. Available at: <https://www.hammerdb.com/> (Accessed 03 February 2020).
- [8] Microsoft Corporation, (2018). *Create Indexed Views*. Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/views/create-indexed-views?view=sql-server-ver15> (Accessed 21 July 2020).
- [9] Oracle Corporation, (2018). *Advanced Materialized Views*. Available at: <https://docs.oracle.com/database/121/DWHSYG/advmv.htm#DWHSYG-GUID-F7394DFE-7CF6-401C-A312-C36603BEB01B> (Accessed 21 July 2020).
- [10] Microsoft Corporation (2017). ‘Configure Parallel Index Operations’. Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/indexes/configure-parallel-index-operations?view=sql-server-ver15> (Accessed 21 June 2020).

Chapter 9

- [1] Knuth, D. (1965). ‘On the translation of languages from left to right’. *Information and Control*, vol. 8, pp 607-639. Available at: <https://www.sciencedirect.com/science/article/pii/S0019995865904262> (Accessed 06 February 2019).
- [2] Atzeni, P., Jensen, C.S., Orsi, G., Ram, S., Tanca, L. and Torlone, R. (2013). ‘The relational model is dead, SQL is dead, and I don't feel so good myself’. *ACM SIGMOD Record*, 42(2), pp.64-68.
- [3] Ireland, C., Bowers, D., Newton, M. and Waugh, K. (2009). ‘A classification of object-relational impedance mismatch’. *First International Conference on Advances in Databases, Knowledge, and Data Applications*, pp. 36-43. Available at: <https://ieeexplore.ieee.org/abstract/document/5071809> (Accessed 18 November 2020).
- [4] Kimball, R. and Ross, M. (2013). *The Data Warehouse Toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons.
- [5] Codd, E.F. (1971). ‘A data base sublanguage founded on the relational calculus’. *Proceedings of the 1971 ACM SIGFIDET Workshop on Data Description, Access and Control*, pp. 35-68.
- [6] Chen, A. N. (1999). *Improving database performances in a changing environment with uncertain and dynamic information demand: An intelligent database system approach*. Doctoral dissertation. University of Connecticut. Available at: <https://opencommons.uconn.edu/dissertations/AAI9942566/> (Accessed 02 February 2017).

Appendix A – Practitioner Survey Structure

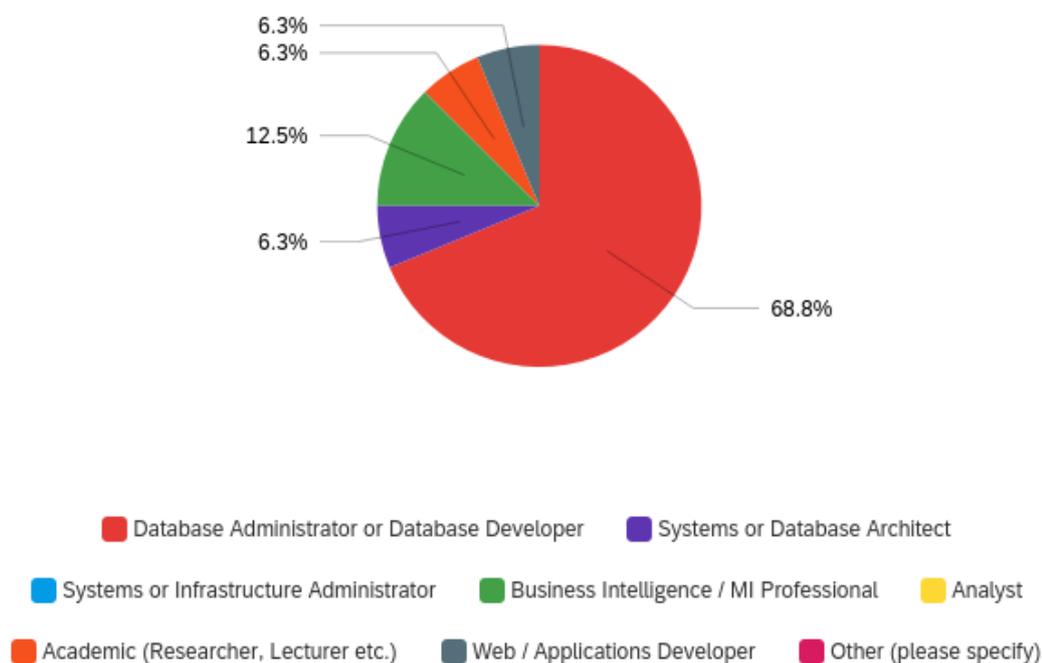
Questions and responses to the survey (n=19) follow below.

Further information can be found in the Qualtrics link here:

http://staffordshire.eu.qualtrics.com/jfe/form/SV_51kxz9na13U8hBX

DB Attitudes Survey

Q1 - How would you describe your primary job role?



| # | Field | Minimum | Maximum | Mean | Std Deviation | Variance | Count |
|---|---|---------|---------|------|---------------|----------|-------|
| 1 | How would you describe your primary job role? - Selected Choice | 1.00 | 7.00 | 2.13 | 1.93 | 3.73 | 16 |

| # | Answer | % | Count |
|---|--|--------|-------|
| 1 | Database Administrator or Database Developer | 68.75% | 11 |
| 2 | Systems or Database Architect | 6.25% | 1 |

| | | | |
|---|---|--------|----|
| 3 | Systems or Infrastructure Administrator | 0.00% | 0 |
| 4 | Business Intelligence / MI Professional | 12.50% | 2 |
| 5 | Analyst | 0.00% | 0 |
| 6 | Academic (Researcher, Lecturer etc.) | 6.25% | 1 |
| 7 | Web / Applications Developer | 6.25% | 1 |
| 8 | Other (please specify) | 0.00% | 0 |
| | Total | 100% | 16 |

Q2 - How many years of experience do you have in your primary job role or function?

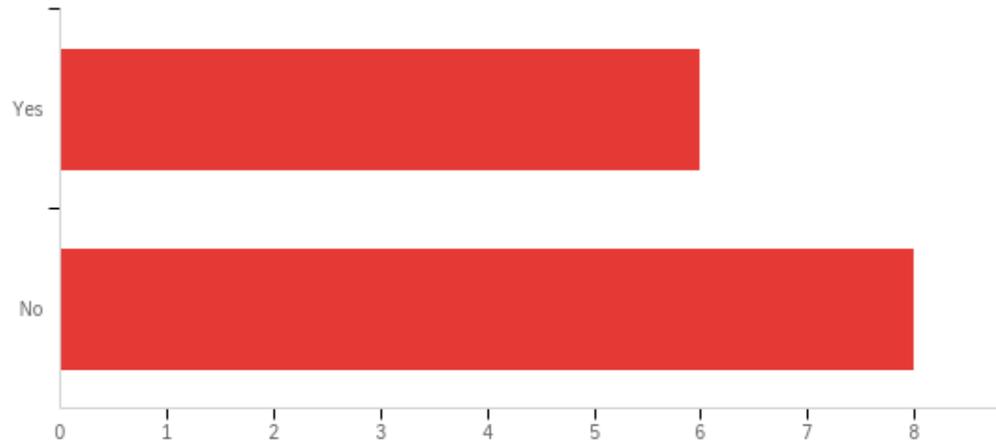


| # | Field | Minimum | Maximum | Mean | Std Deviation | Variance | Count |
|---|--|---------|---------|------|---------------|----------|-------|
| 1 | How many years of experience do you have in your primary job role or function? | 2.00 | 6.00 | 4.75 | 1.39 | 1.94 | 16 |

| # | Answer | % | Count |
|---|-----------------------|--------|-------|
| 1 | Up to 1 year | 0.00% | 0 |
| 2 | Between 1 and 3 years | 12.50% | 2 |

| | | | |
|---|------------------------|--------|----|
| 3 | Between 3 and 5 years | 6.25% | 1 |
| 4 | Between 5 and 7 years | 18.75% | 3 |
| 5 | Between 7 and 10 years | 18.75% | 3 |
| 6 | More than 10 years | 43.75% | 7 |
| | Total | 100% | 16 |

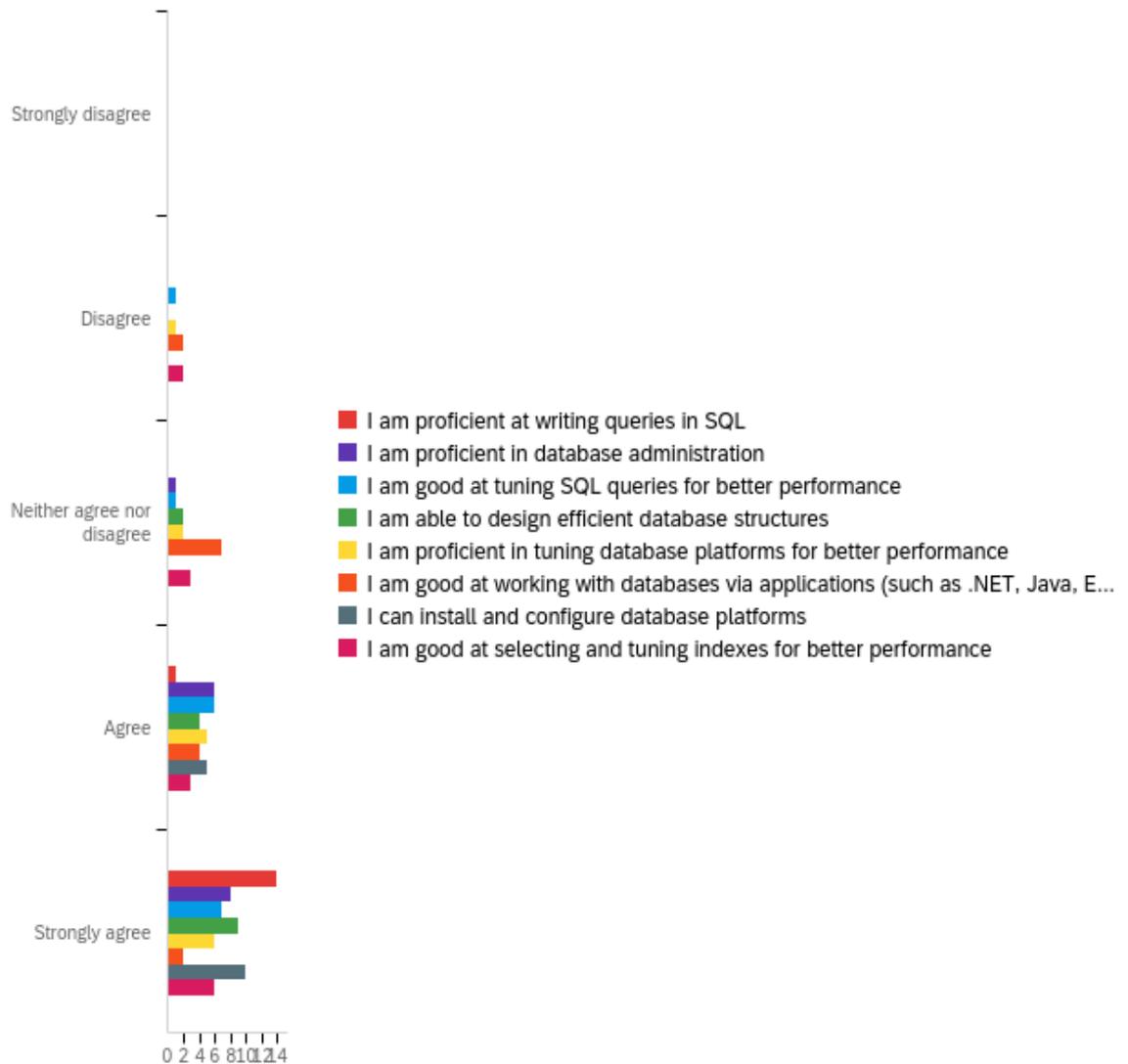
Q25 - In your role, do you use object-relational mapping (ORM) frameworks, or work with databases that process queries generated from ORMs? Examples of ORMs include: Entity Framework, Hibernate / nHibernate; Dapper, ActiveJPA, Enterprise JavaBeans, LINQ to SQL, DataObjects.NET and TopLink.



| # | Field | Min | Max | Mean | Std Dev | Variance | Count |
|---|--|------|------|------|---------|----------|-------|
| 1 | In your role, do you use object-relational mapping (ORM) frameworks, or work with databases that process queries generated from ORMs? Examples of ORMs include: Entity Framework, Hibernate / nHibernate; Dapper, ActiveJPA, Enterprise JavaBeans, LINQ to SQL, DataObjects.NET and TopLink. | 1.00 | 2.00 | 1.57 | 0.49 | 0.24 | 14 |

| # | Answer | % | Count |
|---|--------|--------|-------|
| 1 | Yes | 42.86% | 6 |
| 2 | No | 57.14% | 8 |
| | Total | 100% | 14 |

Q5 - Please indicate how much you would agree, or disagree, with the following statements:



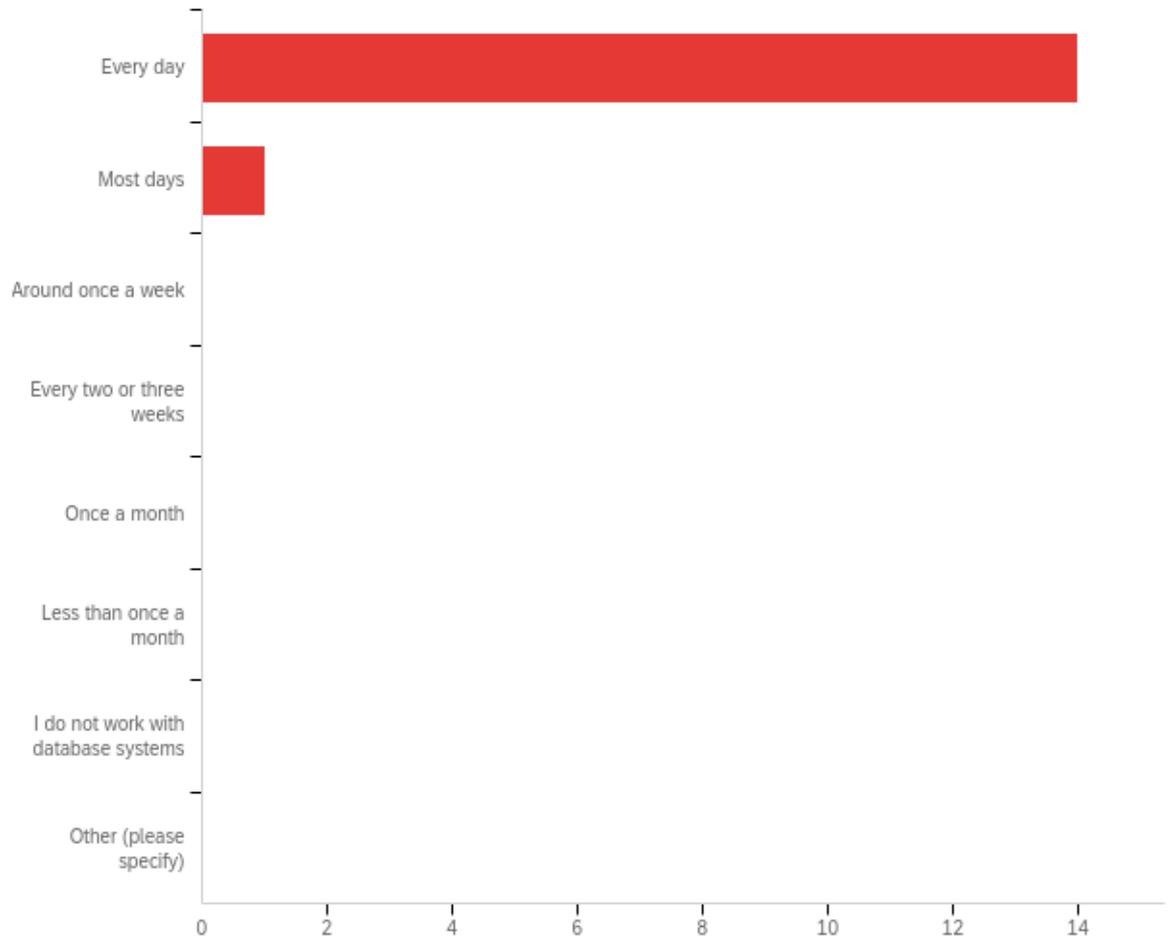
| # | Field | Minimum | Maximum | Mean | Std Deviation | Variance | Count |
|---|---|---------|---------|------|---------------|----------|-------|
| 1 | I am proficient at writing queries in SQL | 6.00 | 7.00 | 6.93 | 0.25 | 0.06 | 15 |

| | | | | | | | |
|---|--|------|------|------|------|------|----|
| 2 | I am proficient in database administration | 4.00 | 7.00 | 6.40 | 0.80 | 0.64 | 15 |
| 3 | I am good at tuning SQL queries for better performance | 2.00 | 7.00 | 6.07 | 1.34 | 1.80 | 15 |
| 4 | I am able to design efficient database structures | 4.00 | 7.00 | 6.33 | 1.01 | 1.02 | 15 |
| 5 | I am proficient in tuning database platforms for better performance | 2.00 | 7.00 | 5.86 | 1.46 | 2.12 | 14 |
| 6 | I am good at working with databases via applications (such as .NET, Java, Excel) | 2.00 | 7.00 | 4.67 | 1.53 | 2.36 | 15 |
| 7 | I can install and configure database platforms | 6.00 | 7.00 | 6.67 | 0.47 | 0.22 | 15 |
| 8 | I am good at selecting and tuning indexes for better performance | 2.00 | 7.00 | 5.43 | 1.80 | 3.24 | 14 |

| # | Question | Strongly disagree | | Disagree | | Neither agree nor disagree | | Agree | | Strongly agree | | Total |
|---|--|-------------------|---|----------|---|----------------------------|---|--------|---|----------------|----|-------|
| 1 | I am proficient at writing queries in SQL | 0.00% | 0 | 0.00% | 0 | 0.00% | 0 | 6.67% | 1 | 93.33% | 14 | 15 |
| 2 | I am proficient in database administration | 0.00% | 0 | 0.00% | 0 | 6.67% | 1 | 40.00% | 6 | 53.33% | 8 | 15 |
| 3 | I am good at tuning SQL queries for better performance | 0.00% | 0 | 6.67% | 1 | 6.67% | 1 | 40.00% | 6 | 46.67% | 7 | 15 |

| | | | | | | | | | | | | |
|---|--|-------|---|--------|---|--------|---|--------|---|--------|----|----|
| 4 | I am able to design efficient database structures | 0.00% | 0 | 0.00% | 0 | 13.33% | 2 | 26.67% | 4 | 60.00% | 9 | 15 |
| 5 | I am proficient in tuning database platforms for better performance | 0.00% | 0 | 7.14% | 1 | 14.29% | 2 | 35.71% | 5 | 42.86% | 6 | 14 |
| 6 | I am good at working with databases via applications (such as .NET, Java, Excel) | 0.00% | 0 | 13.33% | 2 | 46.67% | 7 | 26.67% | 4 | 13.33% | 2 | 15 |
| 7 | I can install and configure database platforms | 0.00% | 0 | 0.00% | 0 | 0.00% | 0 | 33.33% | 5 | 66.67% | 10 | 15 |
| 8 | I am good at selecting and tuning indexes for better performance | 0.00% | 0 | 14.29% | 2 | 21.43% | 3 | 21.43% | 3 | 42.86% | 6 | 14 |

Q6 - How often, on average, do you use, administer or otherwise work with database systems?

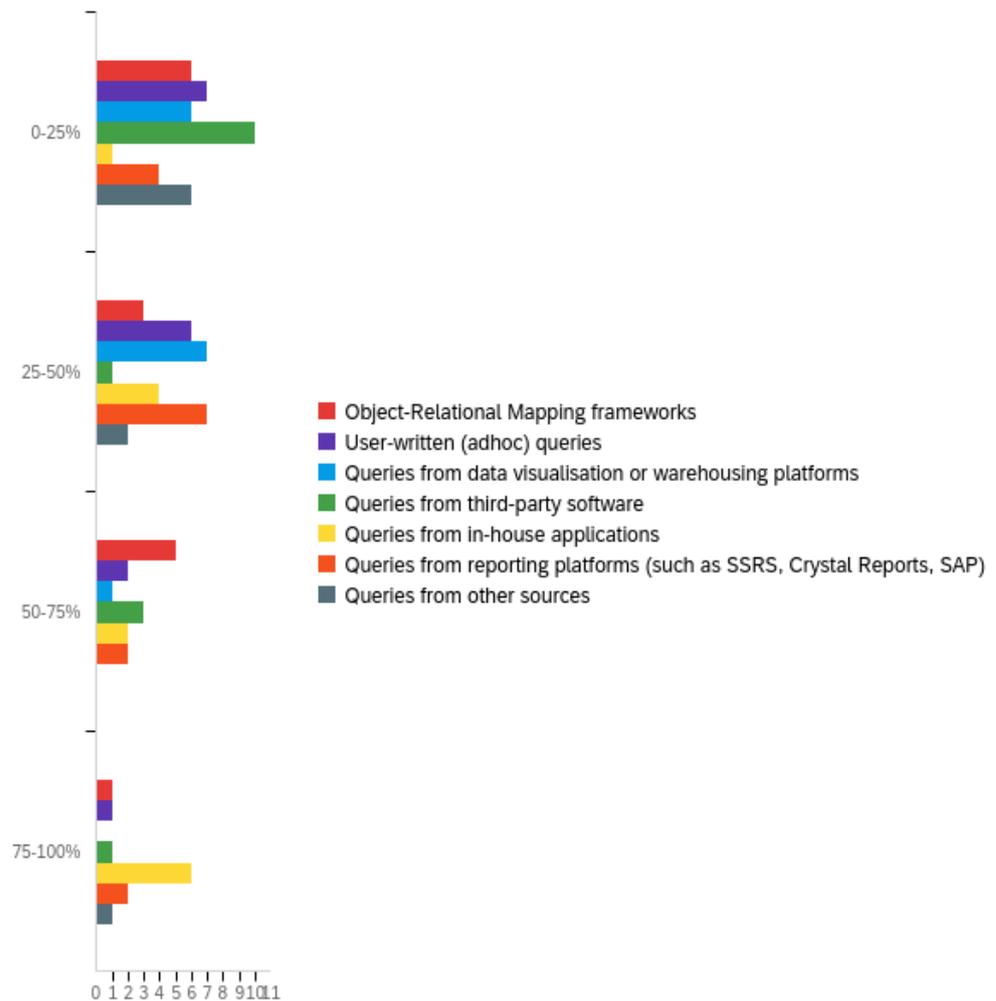


| # | Field | Minimum | Maximum | Mean | Std Deviation | Variance | Count |
|---|--|---------|---------|------|---------------|----------|-------|
| 1 | How often, on average, do you use, administer or otherwise work with database systems? - Selected Choice | 1.00 | 2.00 | 1.07 | 0.25 | 0.06 | 15 |

| # | Answer | % | Count |
|---|--------------------|--------|-------|
| 1 | Every day | 93.33% | 14 |
| 2 | Most days | 6.67% | 1 |
| 3 | Around once a week | 0.00% | 0 |

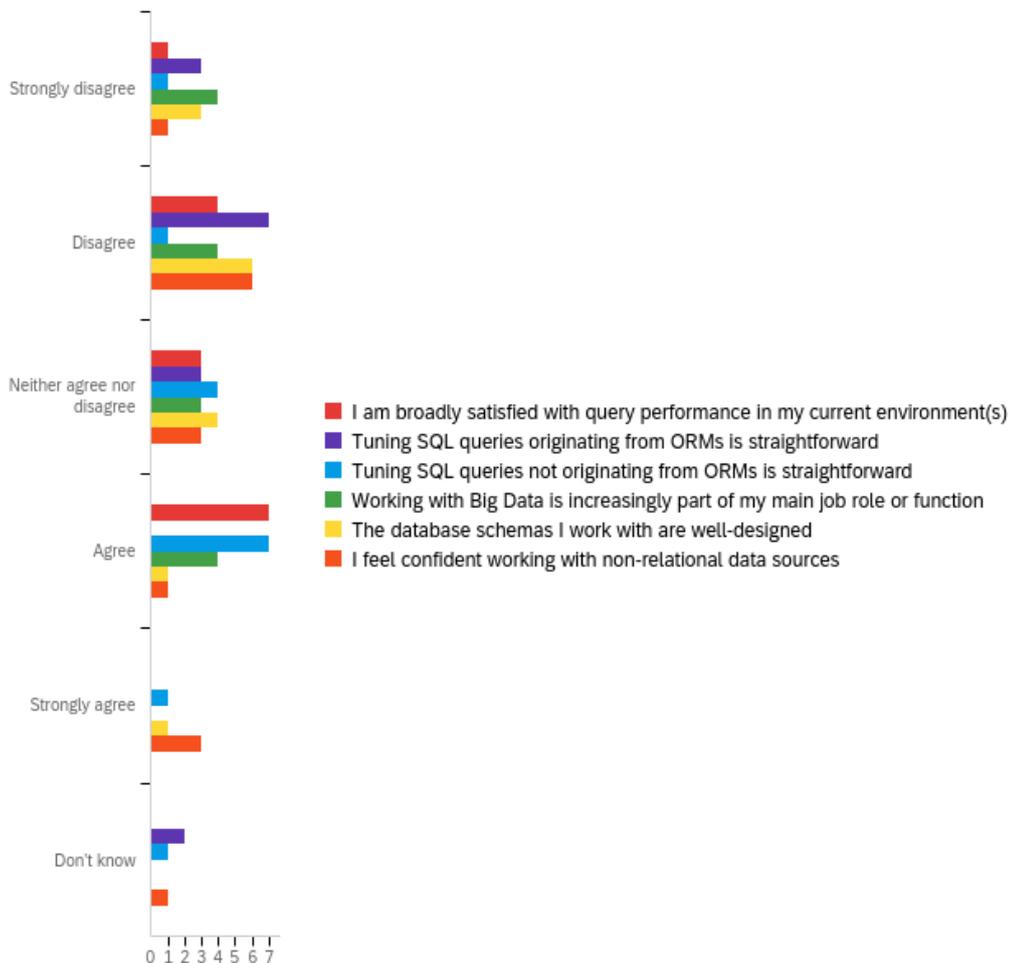
| | | | |
|---|-------------------------------------|-------|----|
| 4 | Every two or three weeks | 0.00% | 0 |
| 5 | Once a month | 0.00% | 0 |
| 6 | Less than once a month | 0.00% | 0 |
| 7 | I do not work with database systems | 0.00% | 0 |
| 8 | Other (please specify) | 0.00% | 0 |
| | Total | 100% | 15 |

Q7 - Thinking about the database platforms that you use or administer the most, please estimate the sources of query traffic:



| # | Question | 0-25% | | 25-50% | | 50-75% | | 75-100% | | Total |
|---|---|--------|----|--------|---|--------|---|---------|---|-------|
| 1 | Object-Relational Mapping frameworks | 40.00% | 6 | 20.00% | 3 | 33.33% | 5 | 6.67% | 1 | 15 |
| 2 | User-written (adhoc) queries | 43.75% | 7 | 37.50% | 6 | 12.50% | 2 | 6.25% | 1 | 16 |
| 3 | Queries from data visualisation or warehousing platforms | 42.86% | 6 | 50.00% | 7 | 7.14% | 1 | 0.00% | 0 | 14 |
| 4 | Queries from third-party software | 66.67% | 10 | 6.67% | 1 | 20.00% | 3 | 6.67% | 1 | 15 |
| 5 | Queries from in-house applications | 7.69% | 1 | 30.77% | 4 | 15.38% | 2 | 46.15% | 6 | 13 |
| 6 | Queries from reporting platforms (such as SSRS, Crystal Reports, SAP) | 26.67% | 4 | 46.67% | 7 | 13.33% | 2 | 13.33% | 2 | 15 |
| 7 | Queries from other sources | 66.67% | 6 | 22.22% | 2 | 0.00% | 0 | 11.11% | 1 | 9 |

Q8 - Please indicate how much you would agree, or disagree, with the following statements:

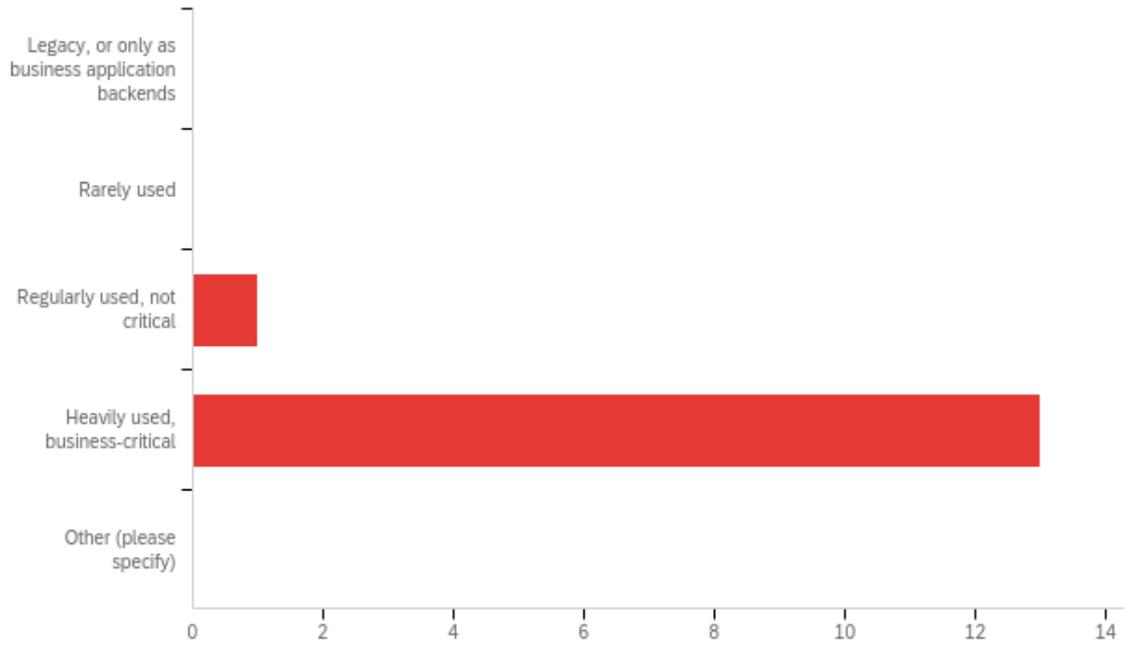


| # | Field | Minimum | Maximum | Mean | Std Deviation | Variance | Count |
|---|--|---------|---------|------|---------------|----------|-------|
| 1 | I am broadly satisfied with query performance in my current environment(s) | 1.00 | 4.00 | 3.07 | 1.00 | 1.00 | 15 |
| 2 | Tuning SQL queries originating from ORMs is straightforward | 1.00 | 6.00 | 2.53 | 1.50 | 2.25 | 15 |
| 3 | Tuning SQL queries not originating from ORMs is straightforward | 1.00 | 6.00 | 3.60 | 1.14 | 1.31 | 15 |
| 4 | Working with Big Data is increasingly part of my main job role or function | 1.00 | 4.00 | 2.47 | 1.15 | 1.32 | 15 |
| 5 | The database schemas I work with are well-designed | 1.00 | 5.00 | 2.40 | 1.08 | 1.17 | 15 |
| 6 | I feel confident working with non-relational data sources | 1.00 | 6.00 | 3.13 | 1.45 | 2.12 | 15 |

| # | Question | Strongly disagree | Disagree | Neither agree nor disagree | Agree | Strongly agree | Don't know | Total |
|---|--|-------------------|-------------|----------------------------|-------------|----------------|-------------|-------|
| 1 | I am broadly satisfied with query performance in my current environment(s) | 6.67% 1 | 26.67% 4 | 20.00% 3 | 46.67% 7 | 0.00% 0 | 0.00% 0 | 15 |
| 2 | Tuning SQL queries | 20.00% 3 | 46.67% 7 | 20.00% 3 | 0.00% 0 | 0.00% 0 | 13.33% 2 | 15 |

| | | | | | | | | | | | | | | |
|---|--|--------|---|--------|---|--------|---|--------|---|--------|---|-------|---|----|
| | originating from ORMs is straightforward | | | | | | | | | | | | | |
| 3 | Tuning SQL queries not originating from ORMs is straightforward | 6.67% | 1 | 6.67% | 1 | 26.67% | 4 | 46.67% | 7 | 6.67% | 1 | 6.67% | 1 | 15 |
| 4 | Working with Big Data is increasingly part of my main job role or function | 26.67% | 4 | 26.67% | 4 | 20.00% | 3 | 26.67% | 4 | 0.00% | 0 | 0.00% | 0 | 15 |
| 5 | The database schemas I work with are well-designed | 20.00% | 3 | 40.00% | 6 | 26.67% | 4 | 6.67% | 1 | 6.67% | 1 | 0.00% | 0 | 15 |
| 6 | I feel confident working with non-relational data sources | 6.67% | 1 | 40.00% | 6 | 20.00% | 3 | 6.67% | 1 | 20.00% | 3 | 6.67% | 1 | 15 |

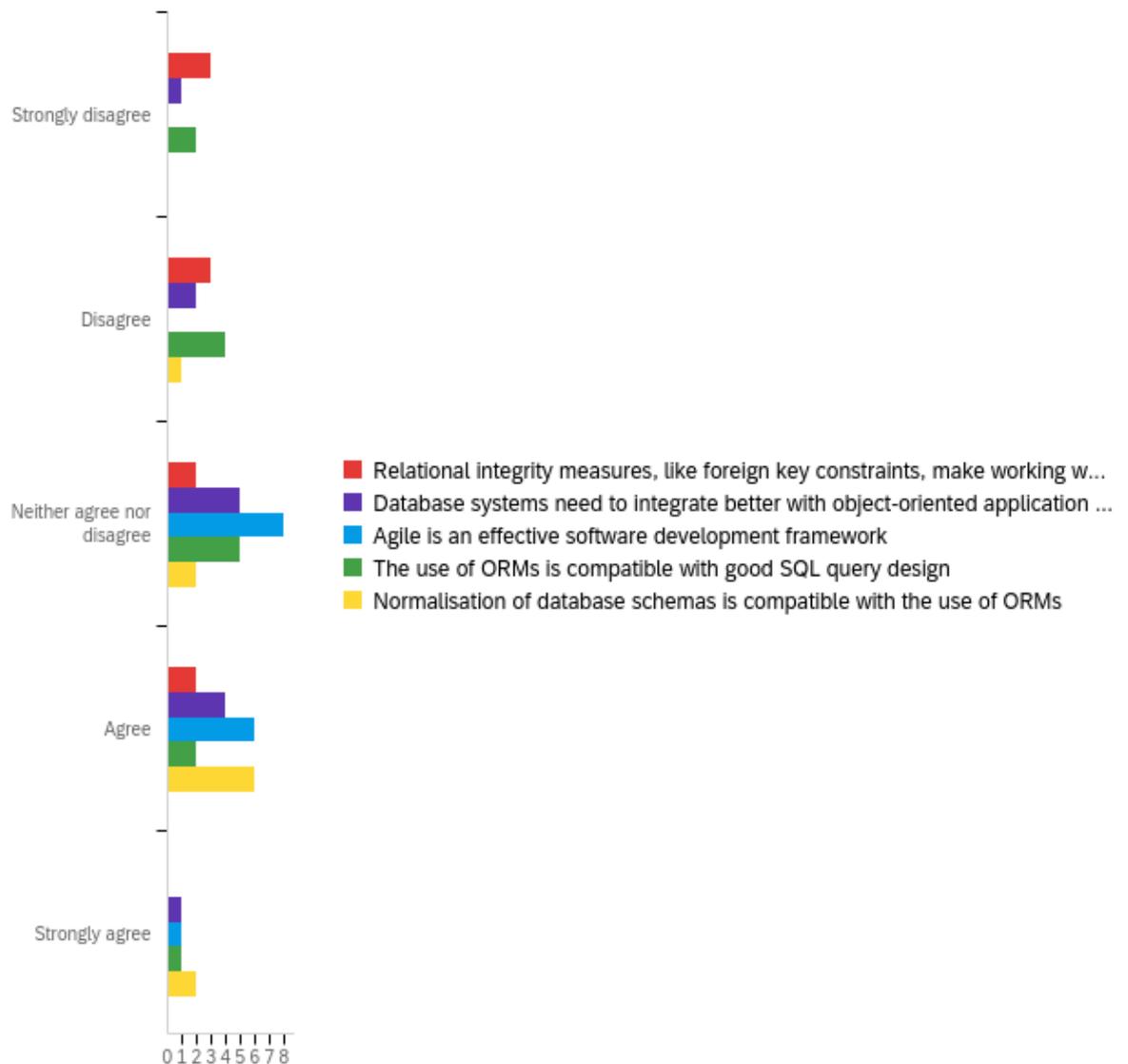
Q26 - How relevant do you believe relational databases will be to organisations in the future?



| # | Field | Minimum | Maximum | Mean | Std Deviation | Variance | Count |
|---|--|---------|---------|------|---------------|----------|-------|
| 1 | How relevant do you believe relational databases will be to organisations in the future? - Selected Choice | 3.00 | 4.00 | 3.93 | 0.26 | 0.07 | 14 |

| # | Answer | % | Count |
|---|--|--------|-------|
| 1 | Legacy, or only as business application backends | 0.00% | 0 |
| 2 | Rarely used | 0.00% | 0 |
| 3 | Regularly used, not critical | 7.14% | 1 |
| 4 | Heavily used, business-critical | 92.86% | 13 |
| 5 | Other (please specify) | 0.00% | 0 |
| | Total | 100% | 14 |

Q21 - Please indicate how much you would agree, or disagree, with the following statements:



| # | Field | Minimum | Maximum | Mean | Std Deviation | Variance | Count |
|---|---|---------|---------|------|---------------|----------|-------|
| 1 | Relational integrity measures, like foreign key constraints, make working with databases harder when using ORMs | 1.00 | 4.00 | 2.30 | 1.10 | 1.21 | 10 |
| 2 | Database systems need to integrate better with object-oriented application development methods | 1.00 | 5.00 | 3.15 | 1.03 | 1.05 | 13 |

| | | | | | | | |
|---|--|------|------|------|------|------|----|
| 3 | Agile is an effective software development framework | 3.00 | 5.00 | 3.53 | 0.62 | 0.38 | 15 |
| 4 | The use of ORMs is compatible with good SQL query design | 1.00 | 5.00 | 2.71 | 1.10 | 1.20 | 14 |
| 5 | Normalisation of database schemas is compatible with the use of ORMs | 2.00 | 5.00 | 3.82 | 0.83 | 0.69 | 11 |

| # | Question | Strongly disagree | | Disagree | | Neither agree nor disagree | | Agree | | Strongly agree | | Total |
|---|---|-------------------|---|----------|---|----------------------------|---|--------|---|----------------|---|-------|
| 1 | Relational integrity measures, like foreign key constraints, make working with databases harder when using ORMs | 30.00% | 3 | 30.00% | 3 | 20.00% | 2 | 20.00% | 2 | 0.00% | 0 | 10 |
| 2 | Database systems need to integrate better with object-oriented application development methods | 7.69% | 1 | 15.38% | 2 | 38.46% | 5 | 30.77% | 4 | 7.69% | 1 | 13 |
| 3 | Agile is an effective software development framework | 0.00% | 0 | 0.00% | 0 | 53.33% | 8 | 40.00% | 6 | 6.67% | 1 | 15 |
| 4 | The use of ORMs is | 14.29% | 2 | 28.57% | 4 | 35.71% | 5 | 14.29% | 2 | 7.14% | 1 | 14 |

| | | | | | | | | | | | | |
|---|--|-------|---|-------|---|--------|---|--------|---|--------|---|----|
| | compatible with good SQL query design | | | | | | | | | | | |
| 5 | Normalisation of database schemas is compatible with the use of ORMs | 0.00% | 0 | 9.09% | 1 | 18.18% | 2 | 54.55% | 6 | 18.18% | 2 | 11 |

Q9 - When working with ORM tools (from any perspective), what are the most regular performance-related challenges that you experience? Please give as much detail as possible.

When working with ORM tools (from any perspective), what are the most regular performance-related challenges that you experience? Please give as much detail as possible.

Overly generic SQL. None of the wider application knowledge is available to the ORM.

Bad queries generated specifying many columns when only one or two needed

SELECT *

N/A

I personally don't use ORM tools in my job.

ORMs tend to generate queries that "work", but are not so great when it comes to efficiency or legibility. It's often difficult to tune those queries to run in a more efficient manner without removing the benefits of using an ORM in the first place. We often run into some issues with parameter sniffing or horrible generated queries/plans as a result of the ORM, but most of the time the code is acceptable, if not the most efficient.

Reproducing issues, understanding queries, dealing with lazy loading and RBAR operations.

It's been a few years since I worked with an ORM as a developer (I'm a DBA now), but I can't seem to recall any *performance* related challenges. Most of the challenges I ran into were related to development and coding of the application with an ORM.

Poorly written queries

ORM is not a silver bullet and was never intended to solve 100% situations. The challenge is identify thw 10% cases where ORM will be a hinderance and convincing team of it

constant churn in the procedure cache

testing how much data can be put in here. testing how much data can be put in here. testing how much data can be put in here. testing how much data can be put in here.

testing how much data can be put in here. testing how much data can be put in here.
testing how much data can be put in here. testing how much data can be put in here.
testing how much data can be put in here. testing how much data can be put in here.
testing how much data can be put in here. testing how much data can be put in here.
testing how much data can be put in here. testing how much data can be put in here.

Q10 - What do you think are the root cause(s) of any performance problems you have experienced with ORMs? Again, please give as much detail as possible.

What do you think are the root cause(s) of any performance problems you have experienced with ORMs? Again, please give as much detail as possible.

Overly generic SQL. None of the wider application knowledge is available to the ORM.

System generated queries are rarely optimal

Developers (or other ORM users) aren't educated how to use it proficiently. And it does require some knowledge about how it works.

bad query designs

N/A

I personally haven't come across any root causes of performance problems, but I've heard that they select too much data.

The ORM just concentrates on accomplishing the task. For more complex tasks that results in some pretty poor performance with lots of nested queries. Sometimes the performance issues arise from the developers adding "just one more thing" to code on a page, resulting in a large number of connect/disconnect operations for each new query. We had one issue where the actual SQL operations took .5s, but the connect/disconnect activity was taking over 30s. Refactoring that code to do more operations at once, use stored procs to do some of the heavier lifting, and reduce the separate operations made a huge difference. In this case, the issue was the developers trying to be efficient in adding new features, but neglecting the way the code was called behind the scenes.

Lazy loading, select star, and RBAR operations

N/A

Indexes that should be included as part of an ORM implimentation depend on development initiative and skill.

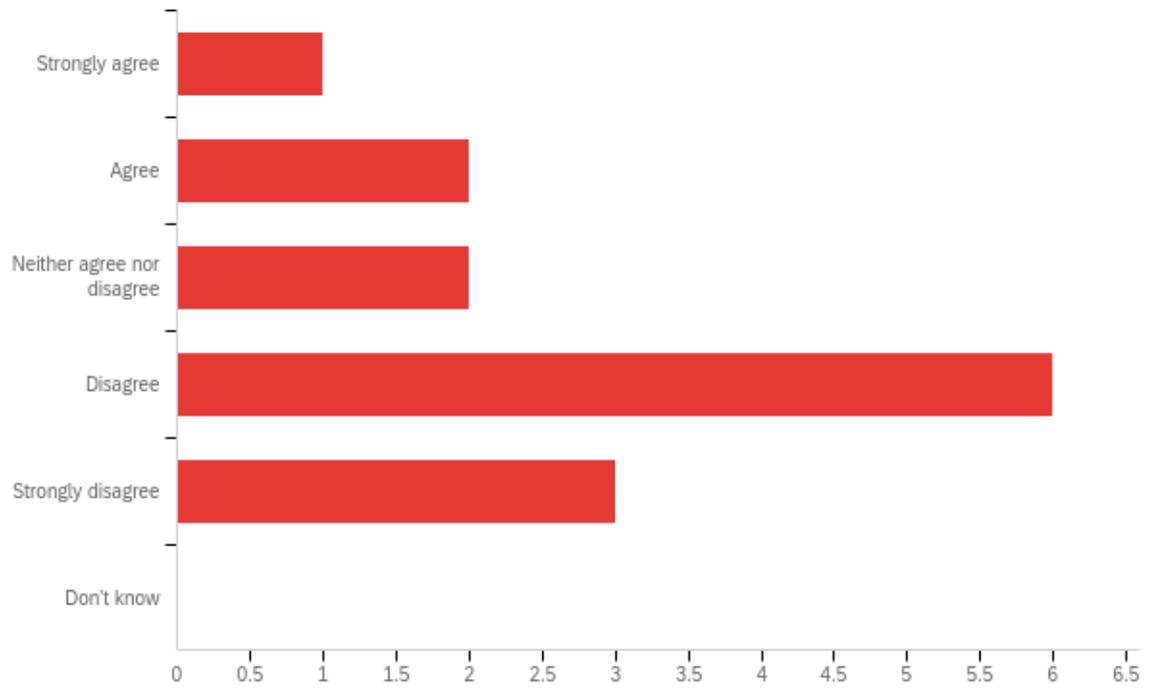
poorly written queries

Bad schema/database/business/architecture design. Trying to hammer ORM to solve a problem ORM is not designed to solve

developpers not understanding how a database works

testing how much data can be put in here. testing how much data can be put in here.
testing how much data can be put in here. testing how much data can be put in here.
testing how much data can be put in here. testing how much data can be put in here.
testing how much data can be put in here.

Q12 - Query performance tuning could be fully replaced by automated processes. Please indicate if you:



| # | Field | Minimum | Maximum | Mean | Std Deviation | Variance | Count |
|---|--|---------|---------|------|---------------|----------|-------|
| 1 | Query performance tuning could be fully replaced by automated processes. Please indicate if you: | 1.00 | 5.00 | 3.57 | 1.18 | 1.39 | 14 |

| # | Answer | % | Count |
|---|----------------------------|--------|-------|
| 1 | Strongly agree | 7.14% | 1 |
| 2 | Agree | 14.29% | 2 |
| 3 | Neither agree nor disagree | 14.29% | 2 |
| 4 | Disagree | 42.86% | 6 |
| 5 | Strongly disagree | 21.43% | 3 |
| 6 | Don't know | 0.00% | 0 |

Q13 - What role could automation have to play in the future of query performance tuning?

What role could automation have to play in the future of query performance tuning?

Agree, but there will be outlying cases the software / ML algorithm cannot comprehend and cater for.

Failing back to a previous query plan when a new one performs badly

It could probably easily work with so called low hanging fruits - queries having obvious issues, like indexing (even if current SQL Server missing index suggestions are bad example for that). Similar with tuning engine configuration to best practices' recommendations. It is still way to go to automated code optimization.

Automatic building of indexes. Better choices by the system when auto-creating statistics and choosing query plans.

Discovering queries that need to be tuned. Automatic tuning most likely won't pick the best option all the time.

We see some of this now - better adapting for query plans, index tuning on the fly, and such. In the future we might see more in-memory indexes, and perhaps even some ways that see patterns and more aggressively cache or aggregate data for retrieval when it comes to those patterns. I think we'll see more proactive alerting on query degradation as well so when queries start performing poorly compared to some automated baseline, the DBAs will get alerts to either act or flag to suspect/re-adjust the baseline.

Guidance towards the right solutions, collecting and analyzing performance data that leads to tuning recommendations

I don't think automation should play any role in query tuning. I feel it is a process best left to manually tuning due to intuition, understanding of and changes to business rules, and getting the most broad coverage of indexes as possible.

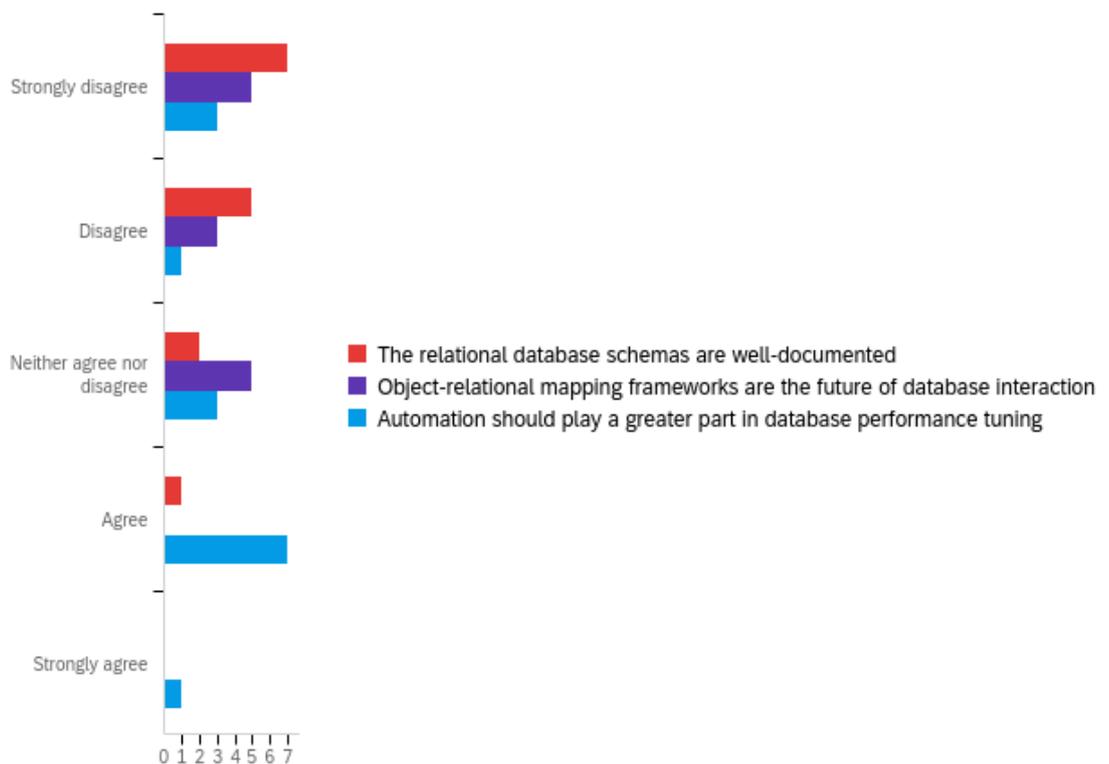
Automated processes can help a lot with performance, in special gathering statistics and using heuristics to point hints, missing indexes, logging errors, etc

don't think it's ever going to happen, need the human touch

Q23 - On a scale of 1 to 10, where 1 represents 'Very unimportant' and 10 represents 'Very important', how important do you think the following concepts are when considering organisational data stores?

| # | Field | Minimum | Maximum | Mean | Std Deviation | Variance | Count |
|---|-----------------|---------|---------|------|---------------|----------|-------|
| 1 | Confidentiality | 1.00 | 10.00 | 8.27 | 2.64 | 7.00 | 15 |
| 2 | Integrity | 6.00 | 10.00 | 9.14 | 1.30 | 1.69 | 14 |
| 3 | Availability | 7.00 | 10.00 | 8.64 | 1.23 | 1.52 | 14 |
| 4 | Flexibility | 3.00 | 10.00 | 6.50 | 1.55 | 2.39 | 14 |
| 5 | Reliability | 4.00 | 10.00 | 8.79 | 1.78 | 3.17 | 14 |
| 6 | Recoverability | 6.00 | 10.00 | 8.93 | 1.39 | 1.92 | 14 |
| 7 | Auditability | 3.00 | 10.00 | 6.62 | 2.10 | 4.39 | 13 |
| 8 | Performance | 5.00 | 10.00 | 8.14 | 1.46 | 2.12 | 14 |

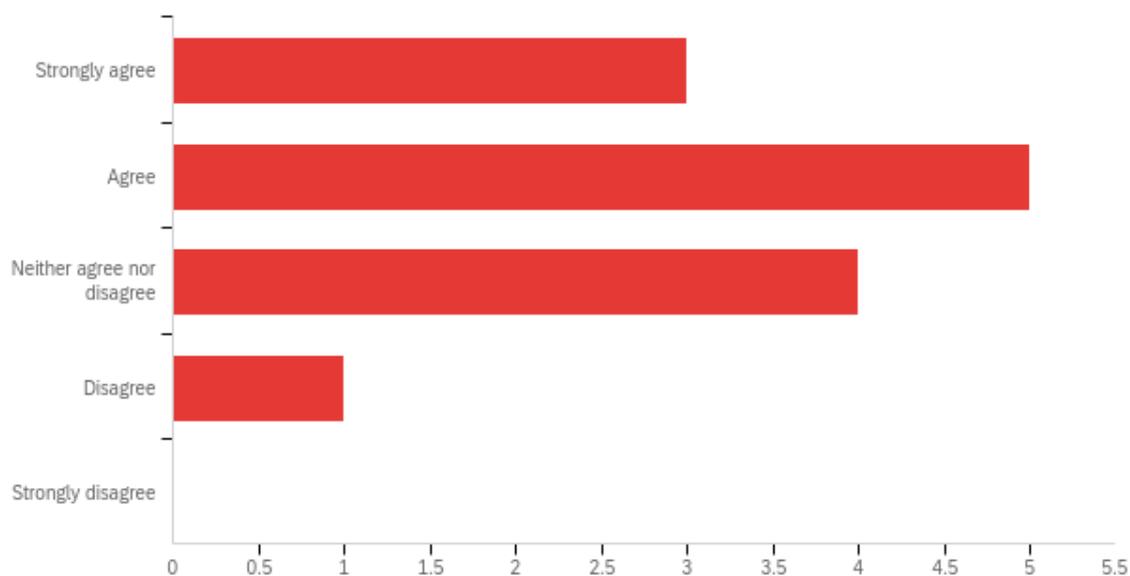
Q16 - Thinking about the database systems that you use or administer the most, how far do you agree or disagree with the following statements?



| # | Field | Minimum | Maximum | Mean | Std Deviation | Variance | Count |
|---|---|---------|---------|------|---------------|----------|-------|
| 1 | The relational database schemas are well-documented | 1.00 | 6.00 | 2.07 | 1.44 | 2.06 | 15 |
| 2 | Object-relational mapping frameworks are the future of database interaction | 1.00 | 4.00 | 2.38 | 1.33 | 1.78 | 13 |
| 3 | Automation should play a greater part in database performance tuning | 1.00 | 7.00 | 4.40 | 2.09 | 4.37 | 15 |

| # | Question | Strongly disagree | | Disagree | | Neither agree nor disagree | | Agree | | Strongly agree | | Total |
|---|---|-------------------|---|----------|---|----------------------------|---|--------|---|----------------|---|-------|
| 1 | The relational database schemas are well-documented | 46.67% | 7 | 33.33% | 5 | 13.33% | 2 | 6.67% | 1 | 0.00% | 0 | 15 |
| 2 | Object-relational mapping frameworks are the future of database interaction | 38.46% | 5 | 23.08% | 3 | 38.46% | 5 | 0.00% | 0 | 0.00% | 0 | 13 |
| 3 | Automation should play a greater part in database performance tuning | 20.00% | 3 | 6.67% | 1 | 20.00% | 3 | 46.67% | 7 | 6.67% | 1 | 15 |

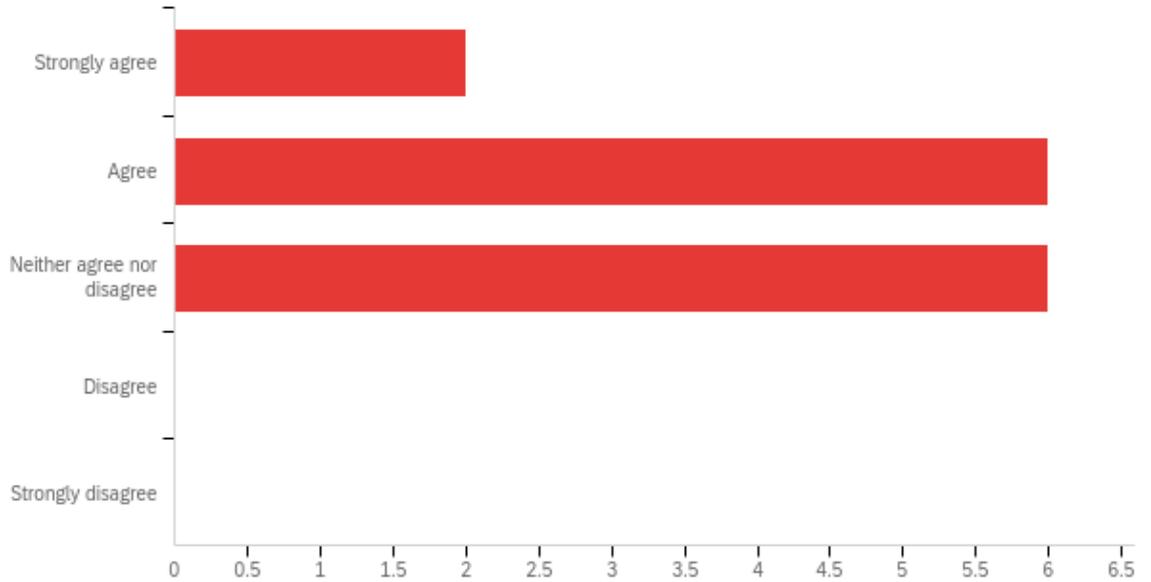
Q22 - Agile development methodologies work well with relational databases. Please indicate if you:



| # | Field | Minimum | Maximum | Mean | Std Deviation | Variance | Count |
|---|--|---------|---------|------|---------------|----------|-------|
| 1 | Agile development methodologies work well with relational databases. Please indicate if you: | 1.00 | 6.00 | 2.69 | 1.49 | 2.21 | 13 |

| # | Answer | % | Count |
|---|----------------------------|--------|-------|
| 1 | Strongly agree | 23.08% | 3 |
| 2 | Agree | 38.46% | 5 |
| 4 | Neither agree nor disagree | 30.77% | 4 |
| 6 | Disagree | 7.69% | 1 |
| 7 | Strongly disagree | 0.00% | 0 |
| | Total | 100% | 13 |

Q24 - Relational databases struggle to perform when dealing with query flows originating from ORM tools. Please indicate if you:



| # | Field | Minimum | Maximum | Mean | Std Deviation | Variance | Count |
|---|--|---------|---------|------|---------------|----------|-------|
| 1 | Relational databases struggle to perform when dealing with query flows originating from ORM tools. Please indicate if you: | 1.00 | 4.00 | 2.71 | 1.16 | 1.35 | 14 |

| # | Answer | % | Count |
|---|----------------------------|--------|-------|
| 1 | Strongly agree | 14.29% | 2 |
| 2 | Agree | 42.86% | 6 |
| 4 | Neither agree nor disagree | 42.86% | 6 |
| 6 | Disagree | 0.00% | 0 |
| 7 | Strongly disagree | 0.00% | 0 |
| | Total | 100% | 14 |

Appendix B: Strong sentiment groupings from interview analysis

Table 1: Strong-sentiment statements from practitioner interviews

| | |
|--------------------------------|---|
| SQL | <p><i>[Interviewer]</i> Do you think SQL was an attractive language in general for application developers?</p> <p><i>[Participant]</i> Yeah, I think so. I think it's, you know, in terms of the structure, it's, it reads like English, which helps, if you, yeah, as long as it's been well formatted, and, and fairly well written economic get the gist of what's happening pretty quickly. There are some terribly written queries and stuff. But it's, I think, I think with the kind of the formatting and the linting and stuff that the different ideas there, it does help if I like when I would get sent a file, or someone would just email over stuff. I just chuck it in for my, to a way that, you know, whenever I would write it, reformat it, so I'm used to reading it.</p> <p><i>[Participant]</i> I think with SQL, if you think about your problem, and you have to say how you wanted to get the data, with a few key words, you can, you can at least make a good attempt at what, what the actual query should be?</p> <p><i>[Participant]</i> ...once they learned the basics of the of the language is easy to get up and running, and at least start pulling down some data, joining pages together, etc. So they can, as analysts, they could get what they need.</p> <p><i>[Participant]</i> Because if I want to do something in SQL, I might need to know, like, 30 commands and total. But if I wanted to do something in [unintelligible] I might need to know, 100, or 200 commands to do.</p> <p><i>[Participant]</i> If somebody asked me, I want, you know, I want to become a data analyst. What kind of tools do I need, what kind of languages so I need to learn? And I would always say SQL is probably the first one because it's quite universal.</p> |
| Query Performance Improvements | <p><i>[Interviewer]</i> Is query performance a really hot topic for the companies you work for? Is it is it you know, a critical thing? Or is it an interesting that the DBA cares about?</p> <p><i>[Participant]</i> Yeah, so probably a lot more of the latter. thing. Yeah. When I, when I joined [company redacted], for example, I was there for two years, a lot of the analysts were already using SQL. And they, one of the DBAs was doing a training session. And it was a kind of a rudimentary intro to SQL for people that had a new SQL before often new analysts come in and learn the database structure. Yeah. And the only thing that he mentioned to do with kind of not locking the database, or any performance was making sure that you added with no lock on joints. And apart from that, there was no other mention of kind of how to optimize or, or anything like that. So you know, for the next two years, any any optimizations, you would have to go and speak to them directly.</p> <p><i>[Participant]</i> ...use kind of distributed servers and clusters and nodes and things like that. It just seems to enhance and processing power. And so try and move everything to a server because when it's all kind of on premises, it feels a lot slower.</p> |

| | |
|---------------------|---|
| | <p><i>[Participant]</i> We were using a sort of a relational database to sort of query and query those 3 million rows, and it just couldn't handle it, it would take two minutes for a report to refresh.</p> <p><i>[Participant]</i> So sometimes stakeholders would be asking questions. And if Alice couldn't get the data within that 20 minute, kind of querying window, then it would be like, I can't analyze your data for the last year, I can only get it for the last week.</p> |
| Query Accessibility | <p><i>[Participant]</i> So yeah, I think having having more training and more knowledge, they would have been able to improve their queries and able to work more efficiently. But at the same time, with the kind of fairly beginner to intermediate knowledge that the analysts have got them, they can get the job done anyway.</p> <p><i>[Participant]</i> I think there should be some kind of, like help with queries, because I think at the moment, it gives you kind of errors, and it will say there was an error on this line, but it doesn't kind of tell you specifically, what caused the error and what the solution is. So I think there should have been kind of like, an error checking sort of algorithm that kind of helps you as you go along with your, of your coding.</p> |
| ORMs | <p><i>[Participant]</i> I think, knowing how helpful ORM is can be in terms of generating the the actual syntax for you.</p> <p><i>[Participant]</i> I've know, kind of no bone to pick with how queries and the way that [ORMs] will structure it, but it's more a case of I'm more more comfortable and familiar with with writing it myself.</p> |
| NoSQL | <p><i>[Participant]</i> I think NoSQL DBs will be the future thing. Databases with designs 50-60 years ago, yeah, the initial concepts. So for stuff that was applicable at the time it, it was good, but with the modern web applications and user interfaces, and just the volume and in different types of data we can collect. Trying to put it all into a SQL DB doesn't make sense when you can have something something like BigQuery or MongoDB, that you can store different types of stuff in there and install get good performance and usage.</p> <p><i>[Participant]</i> See if I've got if I've got audio visual text, on structured text surveys, and kind of standard business operations and orders data, depends on the questions that are coming in, I might have to jump to one dataset or another, and it might be weeks or even months before I go back to something. Yeah. Which point I've often forgotten what the schemer is. And what's, what's the right way?</p> |
| Future of Data | <p><i>[Participant]</i> No, I wouldn't say SQL is [popular]. No, it's not. I mean, it is used but not as, and, but not as popular as I would say it once was.</p> <p><i>[Participant]</i> I would start with SQL, because it's universal, but there's also kind of statistical tools you need to learn like, SAS and R, and Python nowadays, too. So, you know, those are the main sort of languages, I think that anyone would need to learn to become a data [scientist].</p> |

| | |
|-----------------|---|
| | <p><i>[Participant]</i> So if you can imagine, like, for [company redacted], we had 20 million visits to our website every week. And every sort of visit might have, say, 30 interactions. So with data, you know, something like 600 million rows of data every week. And yeah, the way that Google BigQuery was able to kind of process that data was really incredible in comparison to sort of more traditional kind of databases.</p> <p><i>[Participant]</i> I think big data is is like the way forward for almost every kind of solution.</p> <p><i>[Interviewer]</i> Do you think that the DBA is dead yet? Or, they've been predicting it for 50 years, but do you think the DBA is finally ... going to have to upskill or get out?</p> <p><i>[Participant]</i> I think partially, I think what's gonna happen with kind of probably increase in DevOps, increase in data analytics and data science, and then stop being managed on the cloud DBA role, they will become more of the data engineering with BI-type roles.</p> <p><i>[Discussing ability of relational databases to survive]</i></p> <p><i>[Participant]</i> ...Oracle venturing into other areas, you know, I mean? Oh, yes. But as a pure database, it's going to be very difficult.</p> <p><i>[Participant]</i> ...unless something dramatically comes in which takes away databases, then, then maybe, but ... well, you can add things to the you know, you can add, add things, you know, we are beginning to add things to SQL Server. And and I'm just saying that, unless something really dramatically comes and takes away databases, I can't I think they're here to stay.</p> |
| Developers | <p><i>[Participant]</i> I mean, they [the developers] typically work with API's, because they're like, you know, as a digital corporation, [company redacted] a lot of a lot of the data was directly from the web, so and that there was a sort of a mixture.</p> <p><i>[Participant]</i> ...with a lot of the analysts and people I was working with, they weren't particularly [good] with SQL in general in terms of writing their own queries, yeah, and after they would come to me to help them write their own queries.</p> |
| Data Governance | <p><i>[from a conversation about improving the data layer]</i></p> <p><i>[Interviewer]</i> ...you've got an unlimited money magic wand, what would you do?</p> <p><i>[Participant]</i> I first thing I would do from our experience, is make it compulsory that whatever type of database you've got the company invests in, in data dictionaries, or some sort of information based knowledge base on this is what we've got, why we're collecting it, where it's actually coming from, not just one DBA Another DBA that left years ago, yeah, they were getting this, or I remember one, I was looking at different weather data sets that we had on the site. Where's this data coming from? Everyone asked, no one knew? Well, as an analyst, I don't want to trust it. What if it's, yeah, what if it's just someone's someone's made something that's automated? And it's just randomly being generated? It doesn't even match up? So having that a kind of better governance of the data?</p> |

| | |
|----------------------|--|
| | <p><i>[Participant]</i> ...because then I've got the trust in it, because I don't want to use data that I'm not going to trust, or I can't fully, fully account for the kind of the lifeline of where it's come from. Because I'd rather not have the data at all.</p> |
| Data Analysis | <p><i>[Participant]</i> ...as an analyst, I don't care on a row-by-row level, I care on the what's the last 10,000 or 100,000 people that have done this? And what what attributes Do they have that are similar. So I can actually use this data to then market more effectively, or I have a pop up, come up to them on the side. But if it takes 10-15 minutes, you know, they've already left the site.</p> <p><i>[Participant]</i> ...customer's perspective, as you say, they want everything to be to work quickly and efficiently and not have any worries about [that].</p> <p><i>[Participant]</i> ... typically the analytics sandbox, had an update of either every 15 minutes or every hour. So when we were planning, how do we want to do personalization, every we were looking at even every 10 seconds would be too slow. We wanted it ideally every every [sic] second, if possible, which is where kind of ...</p> <p><i>[Interviewer]</i> start running into practical difficulties.</p> <p><i>[Participant]</i> Yes.</p> <p><i>[Participant]</i> ...you have to be extremely efficient at writing queries.</p> <p><i>[Participant]</i> Yeah, I was cuz I still feel like I'm I was thinking about logic in an old sort of sequel [SQL] way. So sometimes when I'm learning new functions and our Python are kind of look at it, and it just feels like really difficult to do the same thing that I wanted to do. And SQL which would, which is really easy to do.</p> <p><i>[Participant]</i> I work with big data sometimes as well. So, and that's how they're I think that's how they're raised. To kind of process, you know, billions and billions of rows of data very efficiently. So when I work with like, Google Cloud is incredible how fast as in comparison to something like Teradata.</p> <p><i>[Participant]</i> I mean, for me, I don't really care where the data comes from neither, whether it's cloud or not.</p> <p><i>[Participant]</i> I just want it [the database] to be available. I don't want it to take hours for me to get the data that I need. Because as an analyst, often you'll be asked a question, and then half an hour later, someone will walk past your desk and the drive go, do you manage to look at that?</p> <p><i>[Participant]</i> ...companies are starting to realize that just collecting and collecting data isn't why the value is the value is going to come from, from actually using it.</p> |
| Cloud Data Analytics | <p><i>[Participant]</i> ...because everything's put everything's in one we see. The beauty is everything's in one area. application. So you, you do the data loading in one place. You can create the tables in one place everything is in a central location, you know, you don't need to go on SQL plus or SQL developer to, you know, to work with the data. You don't need to go on to forms.</p> |

| | |
|-------------------------|--|
| | <p><i>[Participant]</i> But it's having it on the on the cloud with say AWS, it, I think it's more beneficial than is not beneficial. Because you aren't, part of it is you're not really looking, you know, do you know what you want people to know, and you know, what you want people you'd be able to control on what people get to see, but you'd be someone else is going to be looking after that information?</p> |
| <p>3 Vs of Big Data</p> | <p><i>[Participant]</i> So even even [sic] with one month of data, it wasn't really possible to do a join in Tara [Tera]data, then due to the volume of data that was.</p> <p><i>[Interviewer]</i> So that I mean, I might know the answer to this one, then. But would it be fair to say that the relational database system Teradata in this case wasn't necessarily the best the best solution for what you're trying to do there?</p> <p><i>[Participant]</i> Yeah, absolutely. So that that much day, so it wasn't possible to really process and aggregate the data within Teradata.</p> |

Appendix C: Query objectives and code listings from the initial investigation

Table 1: Query Objective O1

| Descriptor | Values |
|---------------|---|
| Summary | Return the mean average air temperature for all buoys on a month-by-month, year-by-year basis, ordered by month and year ascending. |
| Manual SQL | <pre>SELECT [dimDate].[mthNum], [dimDate].[yrNum], AVG([factTAO].[airTemp]) AS [airtemp__avg] FROM [factTAO] INNER JOIN [dimDate] ON ([factTAO].[dateKey] = [dimDate].[dateKey]) GROUP BY [dimDate].[mthNum], [dimDate].[yrNum] ORDER BY [dimDate].[mthNum] ASC, [dimDate].[yrNum] ASC</pre> |
| Python/Django | <pre>FactTAO.objects.all().select_related('datekey').values('datekey__mthnum', 'datekey__yrnum').annotate(Avg('airtemp')).order_by('datekey__mthnum', 'datekey__yrnum')</pre> |
| ORM SQL | <pre>SELECT [dimDate].[mthNum], [dimDate].[yrNum], AVG(CONVERT(float, [factTAO].[airTemp])) AS [airtemp__avg] FROM [factTAO] INNER JOIN [dimDate] ON ([factTAO].[dateKey] = [dimDate].[dateKey]) GROUP BY [dimDate].[mthNum], [dimDate].[yrNum] ORDER BY [dimDate].[mthNum] ASC, [dimDate].[yrNum] ASC</pre> |

Table 2: Query Objective O2

| Descriptor | Values |
|---------------|--|
| Summary | Return the earliest and latest dates for which buoy sensor readings exist within the data set. |
| Manual SQL | <pre>SELECT MIN(f.dateKey) [earliestDate], MAX(f.dateKey) [latestDate] FROM dbo.factTAO f</pre> |
| Python/Django | <pre>FactTAO.objects.aggregate(Min('datekey'), Max('datekey'))</pre> |
| ORM SQL | <pre>SELECT MIN([factTAO].[dateKey]) AS [datekey__min], MAX([factTAO].[dateKey]) AS [datekey__max] FROM [factTAO]</pre> |

Table 3: Query Objective O3

| Descriptor | Values |
|------------|---|
| Summary | Return the latitude and longitude positions of all buoys in January 1984, with no ordering. |
| Manual SQL | <pre>SELECT f.obsID, l.lat, l.long</pre> |

| | |
|---------------|---|
| | <pre> FROM dbo.factTAO f INNER JOIN dbo.dimLocation l ON f.locationKey = l.locationKey INNER JOIN dbo.dimDate d ON f.dateKey = d.dateKey WHERE d.yrNum = 1984 AND d.mthNum = 1 </pre> |
| Python/Django | <pre> FactTAO.objects.select_related('dimlocation__locationkey').all() .select_related('dimdate__datekey').all().values('obsid', 'locationkey__lat', 'locationkey__long').filter(datekey__mthnum = 1, datekey__yrnum = 1984) </pre> |
| ORM SQL | <pre> SELECT [factTAO].[obsID], [dimLocation].[lat], [dimLocation].[long] FROM [factTAO] INNER JOIN [dimLocation] ON ([factTAO].[locationKey] = [dimLocation].[locationKey]) INNER JOIN [dimDate] ON ([factTAO].[dateKey] = [dimDate].[dateKey]) WHERE ([dimDate].[mthNum] = 1 AND [dimDate].[yrNum] = 1984) </pre> |

Table 4: Query Objective O4

| Descriptor | Values |
|---------------|--|
| Summary | Analyse sea surface temperature during the year 1990, and return all rows, including missing data, indicating as anomalous all values where the sea surface temperature is further than 2.5 standard deviations from the average for the year, ordered by date ascending. |
| Manual SQL | <pre> SELECT d.dateKey, f.obsID, f.seaSurfaceTemp, CASE WHEN f.seaSurfaceTemp IS NULL THEN 'Data missing' WHEN ABS(f.seaSurfaceTemp - sd.[avg]) > (2.5 * sd.sd) THEN 'Anomalous' ELSE 'Normal' END [isAnomalous] FROM dbo.factTAO f INNER JOIN dbo.dimDate d ON f.dateKey = d.dateKey CROSS JOIN (SELECT AVG(f.seaSurfaceTemp) [avg], STDEV(f.seaSurfaceTemp) [sd] FROM dbo.factTAO f INNER JOIN dbo.dimDate d ON f.dateKey = d.dateKey WHERE d.yrNum = 1990) sd WHERE d.yrNum = 1990 ORDER BY d.dateKey ASC </pre> |
| Python/Django | <pre> aggs = FactTAO.objects.select_related('datekey').filter(datekey__yrnum = '1990').aggregate(Avg('seasurfacetemp'), StdDev('seasurfacetemp')) outer = FactTAO.objects.select_related('datekey').values('datekey', 'obsid', 'seasurfacetemp', isAnomalous = Case(When(seasurfacetemp = None, then = Value('Data Missing')), default = Value('Normal')), output_field = CharField()).filter(datekey__yrnum = 1990).order_by('datekey') for i in outer: </pre> |

| | |
|---------|---|
| | <pre> if abs((float(i.get('seasurfacetemp') or 0) - aggs.get('seasurfacetemp__avg')) > 2.5 * aggs.get('seasurfacetemp__stddev') and (i.get('isAnomalous') != 'Data Missing'): i['isAnomalous'] = 'Anomalous' </pre> |
| ORM SQL | <pre> (@P1 int) SELECT AVG(CONVERT(float, [factTAO].[seaSurfaceTemp])) AS [seasurfacetemp__avg], STDEVP([factTAO].[seaSurfaceTemp]) AS [seasurfacetemp__stddev] FROM [factTAO] INNER JOIN [dimDate] ON ([factTAO].[dateKey] = [dimDate].[dateKey]) WHERE [dimDate].[yrNum] = @P1 (@P1 nvarchar(24),@P2 nvarchar(12),@P3 int) SELECT [factTAO].[dateKey], [factTAO].[obsID], [factTAO].[seaSurfaceTemp], CASE WHEN [factTAO].[seaSurfaceTemp] IS NULL THEN @P1 ELSE @P2 END AS [isAnomalous] FROM [factTAO] INNER JOIN [dimDate] ON ([factTAO].[dateKey] = [dimDate].[dateKey]) WHERE [dimDate].[yrNum] = @P3 ORDER BY [factTAO].[dateKey] ASC </pre> |

Table 5: Query Objective O5

| Descriptor | Values |
|------------|---|
| Summary | Return the approximate distance in miles between the two buoys that were furthest apart on 01 May 1994, ignoring missing data. |
| Manual SQL | <pre> ;WITH locationData AS (SELECT f.obsID, d.dateKey, l.lat, l.long FROM dbo.factTAO f INNER JOIN dbo.dimLocation l ON f.locationKey = l.locationKey INNER JOIN dbo.dimDate d ON f.dateKey = d.dateKey WHERE d.dateKey = '1994-05-01'), allCombinations AS (SELECT l1.obsID [from], l2.obsID [to], l1.lat [fromLat], l2.lat [toLat], l1.long [fromLong], l2.long [toLong] FROM locationData l1 CROSS JOIN locationData l2), distances AS (SELECT c.[from], c.[to], c.fromLat, c.fromLong, c.toLat, c.toLong, MAX(ACOS(SIN(c.fromLat)*SIN(c.toLat) + COS(c.fromLat)*COS(c.toLat)*COS(c.toLong - c.fromLong)) * 3958.75) [d] FROM allCombinations c GROUP BY c.[from], c.[to], c.[fromLat], c.[toLat], c.fromLong, c.toLong) </pre> |

| | |
|--------------------|---|
| | <pre> SELECT TOP 1 CAST(d.d AS NUMERIC(16,2)) [MaxDistance] FROM distances d ORDER BY [d] DESC </pre> |
| Python / Django | <pre> from django.db.models import Max import math locationData = FactTAO.objects.select_related('datekey', 'locationkey').values('obsid', 'datekey', 'locationkey__lat', 'locationkey__long').filter(datekey = '1994-05-01') locationDataList = list(locationData) vals = [] for i in locationDataList: vals.append(list(i.values())) allCombinations = [] for i in range(0, len(vals)): for j in range(0, len(vals)): r = dict({"from":vals[i][0], "to":vals[j][0], "fromLat":vals[i][2], "toLat":vals[j][2], "fromLong":vals[i][3], "toLong":vals[j][3]}) allCombinations.append(r) for row in allCombinations: LocationDataTempTable(fromField = row.get("from"), toField = row.get("to"), fromLat = row.get("fromLat"), toLat = row.get("toLat"), fromLong = row.get("fromLong"), toLong = row.get("toLong")).save() all = LocationDataTempTable.objects.all() dists = [] for i in all: dists.append(i.distance) max(dists) </pre> |
| ORM SQL | <pre> declare @p1 int set @p1=NULL exec sp_prepexec @p1 output,N'@P1 nvarchar(20)',N'SELECT [factTAO].[obsID], [factTAO].[dateKey], [dimLocation].[lat], [dimLocation].[long] FROM [factTAO] INNER JOIN [dimLocation] ON ([factTAO].[locationKey] = [dimLocation].[locationKey]) WHERE [factTAO].[dateKey] = @P1',N'1994-05-01' select @p1 (the following query is repeated 1,156 times with different parameters) declare @p1 int set @p1=NULL exec sp_prepexec @p1 output,N'@P1 int,@P2 int,@P3 float,@P4 float,@P5 float,@P6 float',N'SET NOCOUNT ON INSERT INTO [locationDataTempTable] ([from], [to], [fromLat], [toLat], [fromLong], [toLong]) VALUES (@P1, @P2, @P3, @P4, @P5, @P6); SELECT CAST(SCOPE_IDENTITY() AS bigint)',997,997,46.064999999999998,46.064999999999998,57.380000000000 0003,57.380000000000003 select @p1 SELECT [locationDataTempTable].[uqid], [locationDataTempTable].[from], [locationDataTempTable].[to], [locationDataTempTable].[fromLat], [locationDataTempTable].[toLat], [locationDataTempTable].[fromLong], [locationDataTempTable].[toLong] FROM [locationDataTempTable] </pre> |

Appendix D: Similarity scoring and schema selection – code listings

This appendix contains the code listings referenced in Chapter 7.

Code Listing 1: Python implementation of Algorithm 1

```
def calculateQuerySimilarity (cubeA, cubeB):
    #calculate Hamming distance
    hamming = 0;
    cubeAEdgeCount = 0;
    cubeBEdgeCount = 0;

    for i in range(0, len(cubeA)):
        for j in range(0, len(cubeA[0])):
            for k in range(0, len(cubeA[0][0])):
                if cubeA[i][j][k] != cubeB[i][j][k]:
                    hamming += 1;
                if cubeA[i][j][k] == 1:
                    cubeAEdgeCount += 1;
                if cubeB[i][j][k] == 1:
                    cubeBEdgeCount += 1;

    maxEdges = max(cubeAEdgeCount, cubeBEdgeCount);

    print("Hamming distance: " + str(hamming));
    print("Maximum number of edges: " + str(maxEdges));

    similarity = round(1.0 - ((hamming / 2.0) / maxEdges),2);
    #print("Query similarity score: " + str(similarity));
    return similarity;

# example wrapper code
def main (sqlQueryA, sqlQueryB):
    import math;
    edgesA = buildEdgeArray (sqlQueryA);
    edgesB = buildEdgeArray (sqlQueryB);
    cubeA = buildAdjacencyCube (edgesA, edgesB, "A");
    cubeB = buildAdjacencyCube (edgesA, edgesB, "B");
    similarity = calculateQuerySimilarity (cubeA, cubeB);
    return similarity;
```

Code Listing 2: Python implementation of Algorithm 2

```
from similarity_functions import * # this is our similarity function code
import psycopg2 # connect to PostgreSQL
import time # standard library

# connect to the PgSQL DB
conn = psycopg2.connect("<credentials>")
testsetdb = conn.cursor()
testsetdb.execute('SELECT rid, stmt, alt FROM testdataraw ORDER BY rid;')
testset = testsetdb.fetchall();

# for each query in the cache (sqlQueryB), run similarity function
for i in testset:
    print 'Processing test query ' + str(i[0]) + '...'
```

```

sqlQueryA = i[1]
querycachedb = conn.cursor()
querycachedb.execute('SELECT queryid, querytextoriginal, queryweight
    FROM querycache ORDER BY queryid;')
querycache = querycachedb.fetchall();
querycachedb.close()
comparison = []
simErrorCount = 0
print 'Assessing similarity of query against cached queries...'
for j in querycache:
    sqlQueryB = j[1]
    try:
        similarity = main(sqlQueryA, sqlQueryB)
    except:
        simErrorCount += 1
        similarity = 0
    similarity = similarity * j[2] # query weight adjustment
    # store query ID and similarity in array
    comparison.append((j[0], similarity))

print 'Total similarity errors: ' + str(simErrorCount)
# lookup k
print 'Fetching k value...' # we do this each time in case K changes
kdb = conn.cursor()
kdb.execute('SELECT k FROM k;')
kval = kdb.fetchall();
for k in kval:
    k = k[0]
kdb.close()

```

Code Listing 3: Finding k-th similar queries to a given query

```

# Fetch k nearest neighbours by similarity
print 'Finding nearest neighbours...'
comparison = sorted(comparison, key=lambda entity: entity[1], reverse=True) # sort
by similarity descending
neighbours = comparison[:int(k)] # slice top k neighbours from list
csv = ''
for n in neighbours:
    csv = csv + str(n[0]) + ', ' # query id
    print 'Identified neighbour: query ' + str(n[0]) + ' with similarity score
' + str(n[1])
csv = csv[:-2] # trim last comma and space
# Fetch majority verdict of schema assignment from neighbours
verdictdb = conn.cursor()
sql = 'SELECT assignedschemaid FROM querycache WHERE queryid IN (' + csv + ')'
verdictdb.execute(sql)
verdict = verdictdb.fetchall()
verdictdb.close()

print 'Finding majority verdict...'

base = 0
alt = 0
for p in verdict:
    if p[0] == 0:
        base += 1
    if p[0] == 1:
        alt += 1

if base >= alt:

```

```

        verdict = 'base'
    if alt > base:
        verdict = 'alt'

    print 'Verdict: Execute against ' + verdict + '.'
    # Start query execution timer
    startTime = time.time()

    # Execute query and return results to caller
    print 'Executing query...'
    sql = ''
    executeWrapper = conn.cursor()
    if verdict == 'base':
        sql = [i[1]]
    if verdict == 'alt':
        sql = [i[2]]
    executeWrapper.callproc('runQuery', sql)
    executeWrapper.execute('commit')
    executeWrapper.close()

    # Stop query execution timer
    stopTime = time.time()
    duration = stopTime - startTime
    print 'Query executed in ' + str(duration) + ' seconds.'

    # Write query, alt query (if applicable), neighbour query IDs and
    # execution time to state table for later async processing
    print 'Writing metadata to state table for asynchronous processing...'
    sql = "INSERT INTO queryqueue SELECT '" + i[1] + "', '" + i[2] + "', " + csv + ",
        " + str(duration)
    sql = [sql]
    addToQueue = conn.cursor()
    addToQueue.callproc('RunQuery', sql)
    addToQueue.execute('commit')
    addToQueue.close()

```

Code Listing 4: Query table definitions

```

CREATE TABLE QueryCache (
    QueryID INTEGER NOT NULL PRIMARY KEY,
    QueryTextOriginal VARCHAR NOT NULL,
    QueryWeight DOUBLE PRECISION,
    AssignedSchemaID INT,
    QueryTextNew VARCHAR,
    LastExecutionDurationSeconds INT
)

CREATE TABLE K (
    K DOUBLE PRECISION
)

CREATE TABLE queryqueue (
    rid INT,
    querytextoriginal VARCHAR(1000),
    querytextnew VARCHAR(1000),
    n1 INT,
    n2 INT,
    n3 INT,
    lastexecutiondurationseconds DOUBLE PRECISION )

```

Code Listing 5: Adjusting query weightings

```
# Open cursor over state table
import psycopg2
import time
from similarity_functions import *
conn = psycopg2.connect("<credentials>")
statetabledb = conn.cursor()

statetabledb.execute("SELECT rid, n1, n2, n3, lastexecutiondurationseconds FROM qu
eryqueue ORDER BY rid ASC;")
statetable = statetabledb.fetchall()
statetabledb.close()

# For each neighbour selected for a query, fetch execution time
qc = conn.cursor()
qc.execute("SELECT k FROM k;")
k = qc.fetchall()
for m in k:
    k = m[0]
k = int(k)

for i in statetable:
    print 'Test query ' + str(i[0]) + ' had execution time : ' + str(i[4]) + '
seconds.'

# Compare against execution time of test query
# If neighbour ran slower, increase weighting by 0.1
# If neighbour ran quicker, decrease weighting by 0.1, vice versa.

    for j in xrange(1, k + 1):
        print j

        reduceWeight = 0
        increaseWeight = 0

        sql = "SELECT lastexecutiondurationseconds FROM querycache WHERE q
ueryid = " + str(i[j])
        qc.execute(sql)
        exectime = qc.fetchall()
        for n in exectime:
            exectime = n[0]

        print 'Top matched query ' + str(i[j]) + ' executed in ' + str(exe
cetime) + ' seconds.'
        print 'Delta: ' + str((float(str(exectime)) - float(str(i[4])))) +
' seconds.'
        if (exectime - i[4]) < 0:
            reduceWeight = 1
            sql = 'UPDATE querycache SET queryweight = queryweight - 0
.1 WHERE queryid = ' + str(i[j])
        if (exectime - i[4]) > 0:
            increaseWeight = 1
            sql = 'UPDATE querycache SET queryweight = queryweight + 0
.1 WHERE queryid = ' + str(i[j])
        print ' '
        sql = [sql]
        qc.callproc('RunQuery', sql)
        qc.execute('commit')
        sql = 'DELETE FROM queryqueue WHERE rid = ' + str(i[0])
        sql = [sql]
        qc.callproc('RunQuery', sql)
```

```

        qc.execute('commit')
        print ' '
qc.close()

```

Code Listing 6: Creating and populating the Chicago data sub-schemas

```

CREATE TABLE chicagoCrimeTypeAlpha (
  rID integer,
  rCaseNumber varchar,
  rDate timestamp,
  rIUCR varchar,
  rPrimaryType varchar,
  rDescription varchar,
  rArrest boolean,
  rDomestic boolean,
  rFBIcode varchar,
  rYear smallint,
  rUpdatedOn timestamp );

CREATE TABLE chicagoCrimeTypeBeta (
  LIKE chicagoCrimeTypeAlpha );

CREATE TABLE chicagoCrimeTypeLocationAlpha (
  rID integer,
  rDate timestamp,
  rBlock varchar,
  rBeat varchar,
  rDistrict varchar,
  rWard integer,
  rCommunityArea varchar,
  rxCoordinate integer,
  ryCoordinate integer,
  rLatitude double precision,
  rLongitude double precision,
  rLocation varchar );

CREATE TABLE chicagoCrimeTypeLocationBeta (
  LIKE chicagoCrimeTypeLocationAlpha );

INSERT INTO chicagoCrimeTypeAlpha
  SELECT rID,
         rCaseNumber,
         rDate,
         rIUCR,
         rPrimaryType,
         rDescription,
         rArrest,
         rDomestic,
         rFBIcode,
         rYear,
         rUpdatedOn
  FROM   chicagoBase
  WHERE  rDate <= ( SELECT MIN(rDate) + (MAX(rDate) - MIN(rDate)) / 2 FROM chicagoBase )

INSERT INTO chicagoCrimeTypeBeta
  SELECT chicagoBase.rID,
         chicagoBase.rCaseNumber,
         chicagoBase.rDate,
         chicagoBase.rIUCR,

```

```

chicagoBase.rPrimaryType,
chicagoBase.rDescription,
chicagoBase.rArrest,
chicagoBase.rDomestic,
chicagoBase.rFBIcode,
chicagoBase.rYear,
chicagoBase.rUpdatedOn
FROM chicagoBase
LEFT JOIN chicagoCrimeTypeAlpha
ON  chicagoBase.rID = chicagoCrimeTypeAlpha.rID
WHERE  chicagoCrimeTypeAlpha.rID IS NULL

INSERT INTO chicagoCrimeLocationAlpha
SELECT rID,
       rDate,
       rBlock,
       rBeat,
       rDistrict,
       rWard,
       rCommunityArea,
       rxCoordinate,
       ryCoordinate,
       rLatitude,
       rLongitude,
       rLocation
FROM  chicagoBase
WHERE rDate <= ( SELECT MIN(rDate) + (MAX(rDate) - MIN(rDate)) / 2 FROM chicagoBase )

INSERT INTO chicagoCrimeLocationBeta
SELECT  chicagoBase.rID,
        chicagoBase.rDate,
        chicagoBase.rBlock,
        chicagoBase.rBeat,
        chicagoBase.rDistrict,
        chicagoBase.rWard,
        chicagoBase.rCommunityArea,
        chicagoBase.rxCoordinate,
        chicagoBase.ryCoordinate,
        chicagoBase.rLatitude,
        chicagoBase.rLongitude,
        chicagoBase.rLocation
FROM  chicagoBase
LEFT JOIN chicagoCrimeLocationAlpha
ON  chicagoBase.rID = chicagoCrimeLocationAlpha.rID
WHERE  chicagoCrimeLocationAlpha.rID IS NULL

```

Code Listing 7: Random SQL query generator

```

SET NOCOUNT ON
GO

DROP PROCEDURE IF EXISTS dbo.chicagoQueryGenerator
GO

CREATE PROCEDURE dbo.chicagoQueryGenerator
AS BEGIN

DECLARE @columnCount TINYINT
DECLARE @counter TINYINT = 0
DECLARE @thisColumn VARCHAR(255)
DECLARE @select VARCHAR(500) = 'SELECT '

```

```

DECLARE @used TABLE ( [name] VARCHAR(255) )

SET          @columnCount = CEILING((
SELECT      TOP 1 c.[column_id]
FROM        sys.columns c
INNER       JOIN sys.tables t ON c.object_id = t.object_id
WHERE       t.[name] = 'chicagobase'
ORDER       BY NEWID() ) / 2.0)

WHILE @counter < @columnCount
BEGIN
    SET @thisColumn = (
        SELECT      TOP 1 c.[name]
        FROM        sys.columns c
        INNER       JOIN sys.tables t ON c.object_id = t.object_id
        LEFT        JOIN @used u ON c.[name] = u.[name]
        WHERE       u.[name] IS NULL
        AND         t.[name] = 'chicagobase'
        ORDER       BY NEWID() )
    INSERT INTO @used VALUES ( @thisColumn )
    SET @select = @select + @thisColumn + ', '
    SET @counter += 1
END
SET          @select = LEFT(@select, LEN(@select) - 1) + ' '

DECLARE @from VARCHAR(500) = ' FROM chicagoBase' + ' '

DECLARE @where VARCHAR(500) = 'WHERE (1=1)' + ' '
-- pick a random number of where clauses, between 0 and 2
DECLARE @numOfWheres TINYINT = ( SELECT ABS(CHECKSUM(NEWID())) % 3 )
DECLARE @colName VARCHAR(255), @dType VARCHAR(255), @val VARCHAR(255)
DECLARE @operator TINYINT, @letters TINYINT
WHILE @numOfWheres > 0
BEGIN
    -- pick a random column from the chicagoBase table
    SELECT @colName = c.[name], @dType = y.[name]
    FROM    sys.columns c
    INNER   JOIN sys.types y ON c.system_type_id = y.system_type_id
    WHERE   c.object_id = OBJECT_ID('chicagoBase')
    AND     c.column_id = ( SELECT ABS(CHECKSUM(NEWID())) %
        ( SELECT COUNT(*) FROM sys.columns c WHERE c.object_id =
OBJECT_ID('chicagoBase') ) + 1 )
    )

    -- now select a random value corresponding to the datatype of the randomly
chosen column
    IF @dType = 'bit' SET @val = CAST(ABS(CHECKSUM(NEWID())) % 2) AS
VARCHAR(255))
    IF @dType LIKE ('%tinyint%') SET @val = CAST(ABS(CHECKSUM(NEWID())) % 255)
AS VARCHAR(255))
    IF @dType = 'datetime' SET @val = '' + CONVERT(VARCHAR,
DATEADD(MINUTE, (ABS(CHECKSUM(NEWID())) % 2629800) * -1, GETDATE()), 120) + '' -- any
time in last 5 years
    IF @dType IN ('decimal', 'numeric', 'float') SET @val =
CAST((ABS(CHECKSUM(NEWID())) % 5000) + ((ABS(CHECKSUM(NEWID())) % 100)/100.0) AS
VARCHAR(255))
    IF @dType IN ('varchar') BEGIN
        SET @val = ''
        SET @letters = ABS(CHECKSUM(NEWID())) % 10 + 1
        WHILE @letters > 0 BEGIN
            SET @val = @val + CHAR(ABS(CHECKSUM(NEWID())) % 26 + 96) --
up to 10 random lowercase ASCII characters
            SET @letters -= 1
        END
    END
END

```

```

        END
        SET @val = '' + @val + ''
    END

    -- construct the WHEREs
    SET @operator = ABS(CHECKSUM(NEWID())) % 4 + 1
    SET @where = @where + 'AND ' + @colName + ' ' +
        CASE    WHEN @operator = 1 THEN '='
                WHEN @operator = 2 AND @val NOT LIKE ('%' + @val + '%') THEN
'>'
                WHEN @operator = 2 AND @val LIKE ('%' + @val + '%') THEN '='
                WHEN @operator = 3 AND @val NOT LIKE ('%' + @val + '%') THEN
'<'
                WHEN @operator = 3 AND @val LIKE ('%' + @val + '%') THEN '='
                WHEN @operator = 4 THEN '!=' END
    SET @where = @where + ' ' + @val + ' '

    SET @numOfWhereas -= 1
END

-- remove the WHERE (1=1) placeholder
IF @where NOT LIKE ('% AND %')
    SET @where = REPLACE(@where, 'WHERE (1=1) ', '')
ELSE
    SET @where = REPLACE(@where, 'WHERE (1=1) AND', 'WHERE')

-- concatenate into a statement
DECLARE @output VARCHAR(1000) = @select + @from + ISNULL(@where, '') + ';'
SELECT @output
END

GO

```

Code Listing 8: Creating query mappings to alternative Chicago sub-schemas

```

-- function to transform a given query on the base schema into a 4-table schema
DECLARE @test VARCHAR(1000) = 'SELECT rLocation, rWard, rDescription, rLatitude FROM
chicagoBase WHERE rIUCR != ''iqfi'' ;'
SELECT * FROM dbo.chicagoQueryTransformer(@test)

DROP FUNCTION IF EXISTS dbo.chicagoQueryTransformer
GO
CREATE FUNCTION dbo.chicagoQueryTransformer ( @inboundQuery VARCHAR(1000) )
RETURNS @outputs TABLE ( inboundQuery VARCHAR(1000), outboundQuery VARCHAR(1000) )
AS BEGIN
    -- use flags to determine which shard and/or partition to use
    DECLARE @typeShardFlag BIT = 0, @locationShardFlag BIT = 0
    DECLARE @alphaPartitionFlag BIT = 0, @betaPartitionFlag BIT = 0
    DECLARE @rDateCount BIGINT, @medianDate DATETIME, @rDate DATETIME
    DECLARE @stringBash VARCHAR(1000), @outboundQuery VARCHAR(1000)

    SET @inboundQuery = REPLACE(@inboundQuery, ';', '') -- causes problems if we don't
remove

    -- determine the shard first
    IF @inboundQuery LIKE ('%rCaseNumber%')
    OR @inboundQuery LIKE ('%rIUCR%')
    OR @inboundQuery LIKE ('%rPrimaryType%')
    OR @inboundQuery LIKE ('%rDescription%')

```

```

OR      @inboundQuery LIKE ('%rArrest%')
OR      @inboundQuery LIKE ('%rDomestic%')
OR      @inboundQuery LIKE ('%rFBIcode%')
OR      @inboundQuery LIKE ('%rUpdatedOn%')
SET @typeShardFlag = 1
IF      @inboundQuery LIKE ('%rBlock%')
OR      @inboundQuery LIKE ('%rBeat%')
OR      @inboundQuery LIKE ('%rDistrict%')
OR      @inboundQuery LIKE ('%rWard%')
OR      @inboundQuery LIKE ('%rCommunityArea%')
OR      @inboundQuery LIKE ('%rxCoordinate%')
OR      @inboundQuery LIKE ('%ryCoordinate%')
OR      @inboundQuery LIKE ('%rLatitude%')
OR      @inboundQuery LIKE ('%rLongitude%')
OR      @inboundQuery LIKE ('%rLocation%')
SET @locationShardFlag = 1

-- now determine the partition, if we can
-- if rDate is present as a predicate in the inbound query, check the median point
-- this will tell us if we can use partition alpha or beta
-- no rDate = both partitions
IF      @inboundQuery LIKE ('%WHERE%rDate%')
BEGIN
    SET @rDateCount = ( SELECT COUNT(*) from dbo.chicagoBase )
    SET @medianDate = (
        SELECT rDate FROM (
            SELECT ROW_NUMBER() OVER ( ORDER BY rDate ASC ) [rid],
rDate
            FROM    dbo.chicagoBase ) median
        WHERE    rid = FLOOR(@rDateCount / 2) )

    -- parse out the date from the inbound query string
    -- bit delicate, this
    SET @stringBash = SUBSTRING(@inboundQuery,
PATINDEX('%WHERE%',@inboundQuery), 1000)
    SET @stringBash = SUBSTRING(@stringBash, PATINDEX('%rDate[<=>! ] [1-2] [0-
9] [0-9] [0-9]%', @stringBash), 38)
    SET @stringBash = REPLACE(@stringBash, ' ', '')
    SET @stringBash = LTRIM(RTRIM(@stringBash))
    SET @stringBash = RIGHT(@stringBash, 19)
    IF ISDATE(@stringBash) = 1
    BEGIN
        SET @rDate = CAST(@stringBash AS DATETIME)
        IF @rDate <= @medianDate
            SET @alphaPartitionFlag = 1
        IF @rDate > @medianDate
            SET @betaPartitionFlag = 1
    END
END

IF      @inboundQuery NOT LIKE ('%WHERE%rDate%')
BEGIN
    SET @alphaPartitionFlag = 1
    SET @betaPartitionFlag = 1
END

-- now glue together a 4-table query based on flag status
--0101 -- location shard, beta partition - no join, no union
--0110 -- location shard, alpha partition - no join, no union
--0111 -- location shard, both partitions - no join, union all
--1001 -- type shard, beta partition - no join, no union
--1010 -- type shard, alpha partition - no join, no union
--1011 -- type shard, both partitions - no join, union all
--1101 -- both shards, beta partition - join on id, no union

```

```

--1110 -- both shards, alpha partition - join on id, no union
--1111 -- all shards and partitions - join on id, union all

    IF @typeShardFlag = 0 AND @locationShardFlag = 1 AND @alphaPartitionFlag = 0 AND
@betaPartitionFlag = 1
        SET @outboundQuery = REPLACE(@inboundQuery, 'chicagoBase',
'chicagoCrimeLocationBeta')
    IF @typeShardFlag = 0 AND @locationShardFlag = 1 AND @alphaPartitionFlag = 1 AND
@betaPartitionFlag = 0
        SET @outboundQuery = REPLACE(@inboundQuery, 'chicagoBase',
'chicagoCrimeLocationAlpha')
    IF @typeShardFlag = 0 AND @locationShardFlag = 1 AND @alphaPartitionFlag = 1 AND
@betaPartitionFlag = 1
        BEGIN
            SET @outboundQuery = REPLACE(@inboundQuery, 'chicagoBase',
'chicagoCrimeLocationAlpha')
            SET @outboundQuery = @outboundQuery + 'UNION ALL ' + CHAR(13) + CHAR(10) +
REPLACE(@outboundQuery, 'chicagoCrimeLocationAlpha',
'chicagoCrimeLocationBeta')
        END
    IF @typeShardFlag = 1 AND @locationShardFlag = 0 AND @alphaPartitionFlag = 0 AND
@betaPartitionFlag = 1
        SET @outboundQuery = REPLACE(@inboundQuery, 'chicagoBase',
'chicagoCrimeTypeBeta')
    IF @typeShardFlag = 1 AND @locationShardFlag = 0 AND @alphaPartitionFlag = 1 AND
@betaPartitionFlag = 0
        SET @outboundQuery = REPLACE(@inboundQuery, 'chicagoBase',
'chicagoCrimeTypeAlpha')
    IF @typeShardFlag = 1 AND @locationShardFlag = 0 AND @alphaPartitionFlag = 1 AND
@betaPartitionFlag = 1
        BEGIN
            SET @outboundQuery = REPLACE(@inboundQuery, 'chicagoBase',
'chicagoCrimeTypeAlpha')
            SET @outboundQuery = @outboundQuery + 'UNION ALL ' + CHAR(13) + CHAR(10) +
REPLACE(@outboundQuery, 'chicagoCrimeTypeAlpha',
'chicagoCrimeTypeBeta')
        END
    IF @typeShardFlag = 1 AND @locationShardFlag = 1 AND @alphaPartitionFlag = 0 AND
@betaPartitionFlag = 1
        SET @outboundQuery = REPLACE(@inboundQuery, 'chicagoBase',
'chicagoCrimeTypeBeta a INNER JOIN chicagoCrimeLocationBeta b ON a.rid =
b.rid')
    IF @typeShardFlag = 1 AND @locationShardFlag = 1 AND @alphaPartitionFlag = 1 AND
@betaPartitionFlag = 0
        SET @outboundQuery = REPLACE(@inboundQuery, 'chicagoBase',
'chicagoCrimeTypeAlpha a INNER JOIN chicagoCrimeLocationAlpha b ON a.rid =
b.rid')
    IF @typeShardFlag = 1 AND @locationShardFlag = 1 AND @alphaPartitionFlag = 1 AND
@betaPartitionFlag = 1
        BEGIN
            SET @outboundQuery = REPLACE(@inboundQuery, 'chicagoBase',
'chicagoCrimeTypeBeta a INNER JOIN chicagoCrimeLocationBeta b ON a.rid =
b.rid')
            SET @outboundQuery = @outboundQuery + 'UNION ALL ' + CHAR(13) + CHAR(10) +
REPLACE(@outboundQuery, 'chicagoCrimeTypeBeta a INNER JOIN
chicagoCrimeLocationBeta b ON a.rid = b.rid',
'chicagoCrimeLocationBeta a INNER JOIN chicagoCrimeLocationBeta b
ON a.rid = b.rid')
        END

-- DEAL WITH THE WHEREAS, APPEARING EACH SIDE OF THE UNION ALLS

-- set up outputs table
INSERT INTO @outputs

```

```

SELECT @inboundQuery, @outboundQuery
RETURN
END

```

Code Listing 9: Example call to generate random SQL queries

```

SET NOCOUNT ON
DECLARE @loopCounter SMALLINT = 1000
DECLARE @results TABLE ( stmt VARCHAR(1000) )
WHILE @loopCounter > 0 BEGIN
    INSERT INTO @results
        EXEC dbo.chicagoQueryGenerator
    SET @loopCounter -= 1
END

DECLARE @allresults TABLE ( stmt VARCHAR(1000), alt VARCHAR(1000) )
INSERT INTO @allresults
    SELECT r.stmt, a.outboundQuery
    FROM @results r
    CROSS APPLY dbo.chicagoQueryTransformer (r.stmt) a

SELECT ROW_NUMBER() OVER ( ORDER BY ( SELECT NULL ) ) [rid], stmt, alt FROM @allresults

```

Code Listing 10: Testing the similarity scoring mechanism

```

testOutcomes = [];

sqlQueryA = "SELECT A.x, B.x FROM A INNER JOIN B ON A.x = B.x;"
sqlQueryB = "SELECT A.x, B.x FROM A INNER JOIN B ON A.x = B.x;"
similarity = main(sqlQueryA, sqlQueryB);
testOutcomes.append(["Set 1", similarity]);

sqlQueryA = "SELECT A.x, B.y FROM B INNER JOIN A ON A.z = B.z;"
sqlQueryB = "SELECT A.x, B.z FROM B INNER JOIN A ON A.z = B.z;"
similarity = main(sqlQueryA, sqlQueryB);
testOutcomes.append(["Set 2", similarity]);

sqlQueryA = "SELECT A.x, A.y, A.z FROM A INNER JOIN B ON A.y = B.y;"
sqlQueryB = "SELECT B.x FROM A INNER JOIN B ON A.y = B.y WHERE A.x = 10;"
similarity = main(sqlQueryA, sqlQueryB);
testOutcomes.append(["Set 3", similarity]);

sqlQueryA = "SELECT A.x FROM A INNER JOIN B ON A.x = B.x WHERE B.y > 100;"
sqlQueryB = "SELECT B.y FROM A INNER JOIN B ON A.z = B.z WHERE A.z = 0;"
similarity = main(sqlQueryA, sqlQueryB);
testOutcomes.append(["Set 4", similarity]);

sqlQueryA = "SELECT A.x FROM A INNER JOIN B ON A.x = B.x WHERE A.x = 10;"
sqlQueryB = "SELECT C.x FROM C INNER JOIN D ON C.z = D.z WHERE D.z > 50;"
similarity = main(sqlQueryA, sqlQueryB);
testOutcomes.append(["Set 5", similarity]);

for i in testOutcomes:
    print(i);

```

Code Listing 11: Testing syntactic and functional validity

```
SET NOCOUNT ON
DECLARE @loopCounter SMALLINT = 100
DECLARE @results TABLE ( stmt VARCHAR(1000) )
WHILE @loopCounter > 0 BEGIN
    INSERT INTO @results
        EXEC dbo.chicagoQueryGenerator
    SET @loopCounter -= 1
END

DECLARE @allresults TABLE ( stmt VARCHAR(1000), alt VARCHAR(1000) )
INSERT INTO @allresults
    SELECT r.stmt, a.outboundQuery
    FROM @results r
    CROSS APPLY dbo.chicagoQueryTransformer (r.stmt) a

DROP TABLE IF EXISTS #finalresults
CREATE TABLE #finalresults ( rid INT, stmt VARCHAR(1000), alt VARCHAR(1000), good BIT )

INSERT INTO #finalresults
    SELECT ROW_NUMBER() OVER ( ORDER BY ( SELECT NULL ) ) [rid],
        stmt, alt, NULL
    FROM @allresults

DECLARE cur_ForEachQueryPair CURSOR LOCAL FAST_FORWARD FOR
    SELECT f.rid, f.stmt, f.alt
    FROM #finalresults f
    WHERE f.alt IS NOT NULL
    ORDER BY rid ASC

DECLARE @thisRid INT, @thisStmt NVARCHAR(1000), @thisAlt NVARCHAR(1000)
DECLARE @badFlag BIT = 0
OPEN cur_ForEachQueryPair
FETCH NEXT FROM cur_ForEachQueryPair INTO @thisRid, @thisStmt, @thisAlt
WHILE @@FETCH_STATUS = 0
BEGIN
    SET @badFlag = 0
    PRINT @thisStmt
    PRINT @thisAlt
    BEGIN TRY
        EXEC sp_executesql @thisStmt
        PRINT 'Good'
    END TRY
    BEGIN CATCH
        SET @badFlag = 1
    END CATCH
    BEGIN TRY
        EXEC sp_executesql @thisAlt
        PRINT 'Good'
    END TRY
    BEGIN CATCH
        SET @badFlag = 1
        PRINT 'Bad'
    END CATCH
    UPDATE #finalResults
    SET good =
        ( SELECT CASE WHEN @badFlag = 0 THEN 1 ELSE 0 END )
    WHERE rid = @thisRid
    FETCH NEXT FROM cur_ForEachQueryPair INTO @thisRid, @thisStmt, @thisAlt
END
CLOSE cur_ForEachQueryPair
DEALLOCATE cur_ForEachQueryPair
```

```
SELECT good, COUNT(*)  
FROM #finalresults f  
GROUP BY good
```

Appendix E: Dynamic schemas – algorithms and code

This material supplements the presentation of the implementation of the dynamic schema redefinition process described in Chapter 8.

E.1 Implementation of the query parser component

Algorithm:

| |
|---|
| <u>Algorithm name:</u> Query parser |
| <u>Inputs:</u> Query plan cache from RDBMS |
| <u>Outputs:</u> Progress log; global temporary tables ##cs and ##q |
| <u>Notes:</u> Progress logging takes place throughout and is omitted for clarity. |

```
Let ##cs be a global temporary table containing cached query statistics.
Insert into ##cs the following values from the query cache, per query:
----|Plan handle,
----|Statement start offset,
----|Statement end offset,
----|Last logical reads,
----|Last elapsed time,
----|Query plan

Let ##q be a global temporary table containing derivatives (D) of query plan cache
details.
Insert into ##q the following values from the query cache, per query:
----|Plan handle,
----|Query text,
----|(D)Selection components of the query text,
----|(D)Non-selection components of the query text occurring after the selection
components,
----|Query use count

Using the query plan cache, update ##q with the latest use counts per query:
----|Joining on plan handle:
----|----|Update use counts in ##q with metric from plan cache.

For each row in ##q:
----|Where the query text is NOT LIKE ('%CREATE%PROCEDURE%') AND
----|Where the query text is NOT LIKE ('%CREATE%VIEW%')
----|----|Derive the attributes, data sources and predicates into separate columns in
##q:
----|----|----|Set ##q.attributes to a derived substring of ##q.attributes:
----|----|----|----|Set ##q.attributes to substring of ##q.attributes from char 1 to:
----|----|----|----|----|CASE WHEN ##q.attributes LIKE ('%FROM%')
----|----|----|----|----|----|Then from 1 to the beginning of the string 'FROM'
----|----|----|----|----|----|Else from 1 to 8000
----|----|----|Set ##q.data_sources to a derived substring of ##q.data_sources:
----|----|----|----|Set ##q.data_sources to substring of ##q.data_sources from char 1 to:
----|----|----|----|----|CASE WHEN ##q.data_sources LIKE ('%WHERE%')
----|----|----|----|----|----|Then from 1 to the beginning of the string 'WHERE'
----|----|----|----|----|----|Else from 1 to 8000
----|----|----|If ##q.predicates NOT LIKE the string ('%WHERE%')
----|----|----|Then set ##q.predicates to an empty string
----|----|----|Else nothing

For all rows in ##q:
----|Where the date_updated is not null AND
```

```
----|Where the date_updated is older than 1 day from today's date
----|----|Delete the row.
```

Code Listing:

```
SET NOCOUNT ON
DECLARE @LogMessage VARCHAR(MAX)

-- Dump the cached stats for later use - to work around cache flush
INSERT INTO ##cs (    plan_handle, statement_start_offset, statement_end_offset,
                    last_logical_reads, last_elapsed_time, query_plan )
    SELECT  s.plan_handle, s.statement_start_offset, s.statement_end_offset,
            s.last_logical_reads, s.last_elapsed_time, q.query_plan
    FROM    sys.dm_exec_query_stats s
    CROSS   APPLY sys.dm_exec_text_query_plan (s.plan_handle,
s.statement_start_offset, s.statement_end_offset) q

-- Scan the plan cache for new queries
INSERT INTO ##q (plan_handle, query_text, attributes, datasources, predicates, usecounts)
    SELECT  cp.plan_handle, t.[text],
            SUBSTRING(t.[text], CHARINDEX('SELECT', t.[text], 1), 8000),
            SUBSTRING((SUBSTRING(t.[text], CHARINDEX('SELECT', t.[text], 1),
8000)), CHARINDEX('FROM', SUBSTRING(t.[text], CHARINDEX('SELECT', t.[text], 1), 8000),
1), 8000),
            SUBSTRING(SUBSTRING((SUBSTRING(t.[text], CHARINDEX('SELECT',
t.[text], 1), 8000)), CHARINDEX('FROM', SUBSTRING(t.[text], CHARINDEX('SELECT', t.[text],
1), 8000), 1), 8000)), CHARINDEX('WHERE', SUBSTRING((SUBSTRING(t.[text],
CHARINDEX('SELECT', t.[text], 1), 8000)), CHARINDEX('FROM', SUBSTRING(t.[text],
CHARINDEX('SELECT', t.[text], 1), 8000), 1), 8000), 1), 8000)),
            cp.usecounts
    FROM    sys.dm_exec_cached_plans cp
    OUTER  APPLY sys.dm_exec_sql_text (cp.plan_handle) t
    LEFT   JOIN ##q q ON cp.plan_handle = q.plan_handle
    WHERE  q.plan_handle IS NULL
    AND    t.[text] NOT LIKE ('%INSERT%')
    AND    t.[text] NOT LIKE ('%UPDATE%')
    AND    t.[text] NOT LIKE ('%DELETE%')
    AND    t.[text] NOT LIKE ('%tpcc_queries%')

SET @LogMessage = 'Inserted ' + CAST(@@ROWCOUNT AS VARCHAR(15)) + ' rows into ##q'
EXEC tpcc_queries.dbo.LogEntry
    @CallingScript = 'QueryParser',
    @CallingCode = 'Scan the plan cache for new queries',
    @LogMessage = @LogMessage

-- Update the queries table with the most current usecounts from the plan cache
UPDATE q
SET    q.usecounts = cp.usecounts,
       q.date_updated = GETDATE()
FROM  ##q q
INNER JOIN sys.dm_exec_cached_plans cp ON q.plan_handle = cp.plan_handle

SET @LogMessage = 'Updated ' + CAST(@@ROWCOUNT AS VARCHAR(15)) + ' rows in ##q'
EXEC tpcc_queries.dbo.LogEntry
    @CallingScript = 'QueryParser',
    @CallingCode = 'Update the queries table with the most current usecounts
from the plan cache',
    @LogMessage = @LogMessage

-- Separate out the top-level attributes, data sources and predicates into separate
columns.
```

```

-- Where this is a CREATE PROCEDURE/VIEW statement (i.e. executing an SP/view), start
from the end of BEGIN.
UPDATE q
SET      q.attributes = SUBSTRING(q.attributes, 1, CASE WHEN q.attributes LIKE
('%FROM%') THEN CHARINDEX('FROM', q.attributes, 1) ELSE 8000 END - 1),
        q.datasources = SUBSTRING(q.datasources, 1, CASE WHEN q.datasources LIKE
('%WHERE%') THEN CHARINDEX('WHERE', q.datasources, 1) ELSE 8000 END - 1),
        q.predicates = CASE WHEN q.predicates NOT LIKE ('%WHERE%') THEN '' ELSE
q.predicates END
FROM    ##q q
WHERE   q.query_text NOT LIKE ('%CREATE%PROCEDURE%')
AND     q.query_text NOT LIKE ('%CREATE%VIEW%')

UPDATE q
SET      q.attributes = SUBSTRING(q.attributes, CHARINDEX('BEGIN', q.attributes,
1) + 5, CASE WHEN q.attributes LIKE ('%FROM%') THEN CHARINDEX('FROM', q.attributes, 1)
ELSE 8000 END - 1),
        q.datasources = SUBSTRING(q.datasources, 1, CASE WHEN q.datasources LIKE
('%WHERE%') THEN CHARINDEX('WHERE', q.datasources, 1) ELSE 8000 END - 1),
        q.predicates = CASE WHEN q.predicates NOT LIKE ('%WHERE%') THEN '' ELSE
q.predicates END
FROM    ##q q
WHERE   q.query_text LIKE ('%CREATE%PROCEDURE%')
OR     q.query_text LIKE ('%CREATE%VIEW%')

-- Delete entries in ##q that haven't been updated in 24 hours.
-- This cascades into the other tables through separate scripts.
DELETE q
FROM    ##q q
WHERE   q.date_updated IS NOT NULL
AND     q.date_updated < DATEADD(DAY, -1, GETDATE())

SET @LogMessage = 'Deleted ' + CAST(@@ROWCOUNT AS VARCHAR(15)) + ' rows from ##q'
EXEC tpcc_queries.dbo.LogEntry
        @CallingScript = 'QueryParser',
        @CallingCode = 'Delete entries in ##q that haven't been updated in 24
hours.',
        @LogMessage = @LogMessage

```

E.2 Temporary table creation

No algorithm supplied (none required).

Code Listing:

```

-- Variant to create if not exists (for SQL Agent job)

IF NOT EXISTS ( SELECT name FROM tempdb.sys.tables WHERE name LIKE ('%##q%') )
BEGIN
    CREATE TABLE ##q
        (
            plan_handle VARBINARY(8000) NOT NULL,
            date_created DATETIME DEFAULT GETDATE() NOT NULL,
            date_updated DATETIME DEFAULT GETDATE(),
            usecounts BIGINT NOT NULL,
            query_text VARCHAR(MAX),
            attributes VARCHAR(MAX),
            datasources VARCHAR(MAX),

```

```

        predicates VARCHAR(MAX),
        suitable_candidate BIT,
        grouped_predicates VARCHAR(MAX),
        CONSTRAINT pk_q_plan_handle PRIMARY KEY (plan_handle)
    )
EXEC tpcc_queries.dbo.LogEntry
    @CallingScript = 'CreateGlobalTemporaryTables',
    @CallingCode = 'CREATE TABLE ##q',
    @LogMessage = 'Created table.'

END

IF NOT EXISTS ( SELECT name FROM tempdb.sys.tables WHERE name LIKE ('%##q_mv_link%') )
BEGIN
    CREATE TABLE ##q_mv_link (
        mv_link_id INT IDENTITY(1,1) NOT NULL,
        plan_handle VARBINARY(8000),
        mv_id INT,
        new_query_text VARCHAR(MAX),
        new_plan_handle VARBINARY(8000),
        date_created DATETIME DEFAULT GETDATE() NOT NULL,
        date_updated DATETIME DEFAULT GETDATE(),
        original_query_cost NUMERIC(24,5),
        original_query_read_count BIGINT,
        original_query_rows BIGINT,
        original_query_columns BIGINT,
        original_query_data_points AS original_query_rows * original_query_columns,
        original_query_efficiency NUMERIC(24,5),
        new_query_cost NUMERIC(24,5),
        new_query_read_count BIGINT,
        new_query_rows BIGINT,
        new_query_columns BIGINT,
        new_query_data_points AS new_query_rows * new_query_columns,
        new_query_efficiency NUMERIC(24,5),
        cost_delta NUMERIC(24,5),
        efficiency_delta NUMERIC(24,5),
        CONSTRAINT mv_link_id PRIMARY KEY (mv_link_id),
        CONSTRAINT fk_plan_handle FOREIGN KEY (plan_handle)
            REFERENCES ##q (plan_handle),
        CONSTRAINT fk_mv_id FOREIGN KEY (mv_id) REFERENCES ##mv (mv_id)
    )
EXEC tpcc_queries.dbo.LogEntry
    @CallingScript = 'CreateGlobalTemporaryTables',
    @CallingCode = 'CREATE TABLE ##q_mv_link',
    @LogMessage = 'Created table.'

END

IF NOT EXISTS ( SELECT name FROM tempdb.sys.tables WHERE name LIKE ('%##mv%') )
BEGIN
    CREATE TABLE ##mv (
        mv_id INT IDENTITY(1,1) PRIMARY KEY NOT NULL,
        associated_view_definition VARCHAR(8000),
        attributes_datasources_predicates
            AS SUBSTRING(associated_view_definition,
                CHARINDEX('AS', associated_view_definition, 1) + 2,
                LEN(associated_view_definition)),
        mv_implemented BIT,
        has_indexed_view BIT
    )
EXEC tpcc_queries.dbo.LogEntry
    @CallingScript = 'CreateGlobalTemporaryTables',
    @CallingCode = 'CREATE TABLE ##mv',
    @LogMessage = 'Created table.'

```

```

END

IF NOT EXISTS ( SELECT name FROM tempdb.sys.tables WHERE name LIKE ('##b%') )
BEGIN
    CREATE TABLE ##b (
        query VARCHAR(8000) )

    EXEC tpcc_queries.dbo.LogEntry
        @CallingScript = 'CreateGlobalTemporaryTables',
        @CallingCode = 'CREATE TABLE ##b',
        @LogMessage = 'Created table.'

END

IF NOT EXISTS ( SELECT name FROM tempdb.sys.tables WHERE name LIKE ('##cs%') )
BEGIN
    CREATE TABLE ##cs (
        plan_handle VARBINARY(64),
        statement_start_offset INT,
        statement_end_offset INT,
        last_logical_reads BIGINT,
        last_elapsed_time BIGINT,
        query_plan NVARCHAR(MAX)
    )

    EXEC tpcc_queries.dbo.LogEntry
        @CallingScript = 'CreateGlobalTemporaryTables',
        @CallingCode = 'CREATE TABLE ##cs',
        @LogMessage = 'Created table.'

END
GO

```

E.3 Implementation of the analyse MVs/use metadata component

Algorithm:

| |
|---|
| Algorithm name: Create and destroy MVs |
| Inputs: Global temporary tables ##q, ##mv, ##mv_link and local temporary table #g |
| Outputs: Local temporary table #g, altered entries in tables ##q, ##mv, ##mv_link, plus MVs to DB |
| Notes: Loops are differentiated from set-based operations by using For each/For all syntax. |
| <pre> Remove own queries from global temporary table ##q: ---- Delete rows from ##q where: ---- ---- Query text LIKE ('##q%') OR ---- ---- Query text LIKE ('##mv%') OR ---- ---- Query text LIKE ('#g) OR ---- ---- Query text LIKE ('%tpcc_queries%') Remove queries with CROSS JOINS from scope: ---- Delete rows from ##q where: ---- ---- Query text includes the string 'CROSS JOIN' Remove MVs no longer present in ##q together with their table entries: ---- Set @count = 0 </pre> |

```

----|For each query q in table ##mv, anti-joining on ##q:
----|----|Replace 'CREATE' with 'DROP', resulting in derived query D(q)
----|----|Replace CR/LF line endings with blank strings
----|----|Increment @count
----|----|Execute the derived query D(q)
----|Delete all rows from ##mv where:
----|----|There exists companion rows in ##mv_link, joining on plan_handle AND
----|----|There does not exist companion rows in ##q, anti-joining on plan handle.
----|Delete all rows from ##mv_link where:
----|----|There do not exist companion rows in ##mv, anti-joining on plan handle.
----|For all rows in ##mv_link:
----|----|Get the max mv_link_id identifier grouped by plan handle and mv_id identifier
----|----|Delete all rows from ##mv_link not in the resulting query set.
----|
Begin parsing/MV creation process:
----|Set variable @c = 0----|
----|Remove nested queries from scope:
----|----|For each row in a subset of table ##q including the following columns:
----|----|Plan handle,
----|----|Use counts,
----|----|Query text,
----|----|Attributes,
----|----|Data sources,
----|----|Predicates----|
----|----|
----|----|Do:
----|----|----|If attributes substring of the row in position 7 to 8000 LIKE '%SELECT%'
----|----|----|OR attributes substring of the row in position 7 to 8000 LIKE '%FROM%'
----|----|----|OR attributes substring of the row in position 7 to 8000 LIKE
'%WHERE%'
----|----|----|Then
----|----|----|Set suitable_candidate attribute of ##q = 0
----|----|----|If data sources substring of the row in position 5 to 8000 LIKE '%SELECT%'
----|----|----|OR data sources substring of the row in position 5 to 8000 LIKE
'%FROM%'
----|----|----|OR data sources substring of the row in position 5 to 8000 LIKE
'%WHERE%'
----|----|----|Then
----|----|----|Set suitable_candidate attribute of ##q = 0
----|----|----|If predicates substring of the row in position 6 to 8000 LIKE '%SELECT%'
----|----|----|OR predicates substring of the row in position 6 to 8000 LIKE '%FROM%'
----|----|----|OR predicates substring of the row in position 6 to 8000 LIKE
'%WHERE%'
----|----|----|Then
----|----|----|Set suitable_candidate attribute of ##q = 0
----|----|
----|----|Remove nested system functions from scope:
----|----|----|If data sources column LIKE ('%(%)%')
----|----|----|Then set suitable_candidate attribute of ##q = 0
----|----|
----|----|Check data sources exist and remove from scope if none:
----|----|----|If data sources column length = 0 or null:
----|----|----|Then set suitable_candidate attribute of ##q = 0
----|----|
----|----|Group identical queries with differing predicates:
----|----|----|If there exist rows in ##q with identical attributes to this attribute AND
----|----|----|If there exist rows in ##q with identical data sources to this data source
AND
----|----|----|If there exist rows in ##q with different predicates to this predicate
----|----|----|Then
----|----|----|Compile this predicate with delimiter | to existing predicates
----|----|----|Update ##q with new predicate value
----|----|
----|----|If the query at hand is not marked as suitable, mark as suitable:

```

```

----|----|----|For the first query in ##q with the same plan handle as the one in hand AND
----|----|----|Where this query has suitable_candidate = 0
----|----|----|Update ##q and set suitable_candidate = 1 for this plan handle.

Define empty variable @predicateList.

For all distinct data sources in ##q:
----|Where predicates exist AND
----|Where suitable_candidate = 1 AND
----|Where predicates are not grouped (grouped_predicates is null)
----|Do:
----|----|For all distinct predicates in ##q:
----|----|----|Where the data source matches the data source at hand AND
----|----|----|Suitable candidate = 1 AND
----|----|----|The predicate is not empty
----|----|Do:
----|----|----|If @predicateList is empty
----|----|----|Then set @predicateList = the distinct predicate at hand
----|----|----|Else append a comma (,) and the distinct predicate at hand to
@predicateList
----|----|
----|----|Set @predicateList = grouped_predicates from ##q
----|----|Where ##q.data_source matches the data source at hand
----|----|And the predicate exists
----|----|And suitable_candidate = 1
----|----|
----|Set @predicateList to an empty string

Remove all queries against system databases from scope:
----|Delete from ##q where:
----|----|Query text LIKE(<name of system database(s) as applicable>)
----|----|*Repeat as necessary
----|----|

If local temporary table #g exists:
----|Drop table #g

Let #g be a local temporary table with columns described as:
----|Plan handle,
----|Attributes,
----|Original attributes,
----|Data sources,
----|Predicates,
----|Original predicates,

Populate #g with all entries from ##q, mapping as follows:
----|#g)Plan handle <- (##q) Plan handle,
----|#g)Attributes <- (##q) Attributes,
----|#g)Original attributes <- (##q) Attributes,
----|#g)Data sources <- (##q) Data sources,
----|#g)Predicates <- (##q) Predicates,
----|#g)Original predicates <- (##q) Predicates
----|Where:
----|----|suitable_candidate = 1

Group queries by same data sources and attributes:
----|Declare empty variable @thisAttributesSplit
----|Declare empty variable @thisAttributes
----|For each distinct data source in #g:
----|----|Create comma-separated list of all #g.attributes, set @thisAttributes to this
list.
----|Deduplicate @thisAttributes:
----|----|For each comma-separated item in @thisAttributes,
----|----|----|Identify first instance of item
----|----|----|Remove all other identical items

```

```

----|----|Set @thisAttributesSplit to the new deduplicated comma-separated list.
----|----|Update #g with attributes = @thisAttributesSplit for the distinct data source at
hand.

For both attribute and original_attribute in #g:
----|Do:
----|Alias each attribute with its own three-part reference:
----|----|For each plan handle, attributes in #g:
----|----|----|For each distinct item in attributes in #g:
----|----|----|----|Replace the word 'SELECT' with an empty string
----|----|----|----|Replace [, ], (, ) characters with an empty string
----|----|----|----|Trim the item to all leftmost characters - 1
----|----|----|----|Add [ ] to the outside of the item
----|----|----|----|Write all items back to the attributes column in #g, CSV separated

For all rows in #g:
----|If data source is not prepended with 'dbo' AND
----|If data source is not a two-part name (contains .)
----|Then update #g, prepending 'dbo.' to data source.
----|

For all rows in #g where query_text contains 'JOIN':
----|If data source is not prepended with 'dbo' AND
----|If data source is not a two-part name (contains .)
----|Then update #g, prepending 'JOIN dbo.' to data source.
----|

For all rows in #g:
----|Replace 'dbo. ' substrings in the data sources column with the string 'dbo.'
----|

For each distinct datasource in #g:
----|Replace all AND with OR (for maximal coverage of conditions)
----|Deduplicate predicates (use same pattern as for data sources)
----|

For all rows in #g:
----|Replace 'WHERE OR' substrings in the predicates column with the string 'OR WHERE'

For all rows in #g:
----|Insert into table ##mv as associated view definition:
----|----|String 'CREATE VIEW <<NEWID>> WITH SCHEMABINDING AS ' (or RDBMS equivalent) PLUS
----|----|g.attributes + ' ' + g.data_sources + ' ' + g.predicates
----|Where:
----|----|The view does not already exist in ##mv
----|----|

Deduplicate ##mv:
----|For all rows, fetch max mv_id, grouping on all non-key columns
----|Delete all rows from ##mv not in this set.
----|

For all rows in ##mv:
----|Update string <<NEWID>> in attributes with NEWID() system function output or
equivalent.
----|

(errata): Fix codepage issues:
----|Replace '&%' patterns with equivalent <, >, <=, >= primitives in ##mv columns.

For all rows in ##mv:
----|Insert into ##mv_link:
----|----|mv.mv_id, g.plan_handle, PLUS
----|----|(g.original_attributes, substring(13 to 37) mv.associated_view_definition PLUS
----|----|g.original_predicates) PLUS
----|----|Current date/time.
----|----|Where:
----|----|----|Link does not currently exist matching this AVD and mv_id.

(errata): Fix double dbo issue
----|Replace 'dbo.dbo' pattern with 'dbo' in all associated_view_definitions

```

```

For all rows in ##mv_link:
----|Set original_query_columns to substrings of query_text, reflecting attributes.

For each MV, check MV parses and executes correctly:
----|For all rows in ##mv
----|Where mv_implemented = 0:
----|----|Construct dynamic CREATE VIEW statement
----|----|Try:
----|----|----|Execute statement
----|----|----|Update mv_implemented = 1 in ##mv
----|----|Catch:
----|----|----|Update mv_implemented = 0 in ##mv
----|----|----|
Materialise the views with clustered indexes:
----|For all rows in ##mv
----|Where has_indexed_view = 0
----|Do:
----|----|Construct string CREATE UNIQUE CLUSTERED INDEX (or equivalent) PLUS
----|----|View name (replacing '-' string with an empty string) PLUS
----|----|'ON' + first column of materialised view.
----|
----|----|Try:
----|----|----|Execute the string.
----|----|----|Update ##mv with has_indexed_view = 1
----|----|Catch:
----|----|----|Construct DROP VIEW statement for view.
----|----|----|Execute DROP VIEW statement.
----|----|----|Construct CREATE TABLE statement replacing view.
----|----|----|Try:
----|----|----|Execute CREATE TABLE statement.
----|----|----|Update has_indexed_view = 1
----|----|----|Catch:
----|----|----|Do nothing.

For all rows in ##mv_link:
----|Add [ ] brackets to predicates in column predicates to avoid parsing issues.
----|
(errata): String replacements of ##mv_link.new_query_text to fix XML codepage issues.

```

Code Listing:

```

USE tpcc
SET NOCOUNT ON
SET QUOTED_IDENTIFIER ON

DECLARE @count INT = 0
DECLARE @LogMessage VARCHAR(MAX)

-- Get the original query cost, the original query read count and original query rows.
Update ##q_mv_link.
DECLARE cur_ForEachQuery CURSOR LOCAL FAST_FORWARD FOR
    SELECT link.mv_link_id
    FROM ##q q
    INNER JOIN ##q_mv_link link ON q.plan_handle = link.plan_handle
    INNER JOIN ##mv mv ON link.mv_id = mv.mv_id
-- WHERE mv.mv_implemented = 1
DECLARE @thisLinkID INT, @thisEstimatedCost NUMERIC(26,10)
DECLARE @results TABLE (
    plan_handle VARBINARY(8000),
    last_logical_reads BIGINT,

```

```

        estimated_cost NUMERIC(26,10),
        estimated_rows FLOAT )

OPEN    cur_ForEachQuery
FETCH  NEXT FROM cur_ForEachQuery INTO @thisLinkID
WHILE  @@FETCH_STATUS = 0
BEGIN
    DELETE FROM @results
        ;WITH XMLNAMESPACES ( DEFAULT
N'http://schemas.microsoft.com/sqlserver/2004/07/showplan' ),
        PlanText AS ( SELECT CAST(cs.query_plan AS XML) AS QueryPlan,
                                cs.last_logical_reads,
                                cs.plan_handle
                                FROM    ##cs cs
                                INNER  JOIN ##q_mv_link link ON cs.plan_handle =
link.plan_handle
                                WHERE  link.mv_link_id = @thisLinkID
        ),
        PlanElements AS (
                                SELECT PlanText.plan_handle,
                                PlanText.QueryPlan,
                                PlanText.last_logical_reads,

                                RelOp.pln.value(N'@EstimatedTotalSubtreeCost', N'float') AS EstimatedCost,
                                RelOp.pln.value(N'@EstimateRows', N'float') AS EstimateRows,
                                RelOp.pln.value(N'@NodeId', N'integer') AS NodeId
                                FROM    PlanText
                                CROSS  APPLY
PlanText.QueryPlan.nodes(N'//RelOp')RelOp(pln)
        )

        INSERT INTO @results ( plan_handle, last_logical_reads, estimated_cost,
estimated_rows )
        SELECT e.plan_handle, e.last_logical_reads, e.EstimatedCost,
e.EstimateRows
        FROM    PlanElements e
        WHERE   e.NodeId = 0

        BEGIN TRY
            UPDATE link
            SET
                link.original_query_cost = CAST(r.estimated_cost AS
NUMERIC(24,5)),
                link.original_query_rows = r.estimated_rows,
                link.original_query_read_count = r.last_logical_reads
            FROM    ##q_mv_link link
            INNER  JOIN @results r ON link.plan_handle = r.plan_handle
            WHERE  link.mv_link_id = @thisLinkID
            AND    link.original_query_cost IS NULL
-- to prevent overwriting previously-captured costs
            OR    link.original_query_rows IS NULL

            SET @count += @@ROWCOUNT

        END TRY
        BEGIN CATCH
            PRINT 'Something went wrong updating the usage metadata.
                Dumping @result to console...'
            SELECT * FROM @results
        END CATCH

        FETCH  NEXT FROM cur_ForEachQuery INTO @thisLinkID
END
CLOSE cur_ForEachQuery
DEALLOCATE cur_ForEachQuery

```

```

SET @LogMessage = 'Updated a total of ' + CAST(@count AS VARCHAR(15)) + ' entries in
##q_mv_link'
EXEC tpcc_queries.dbo.LogEntry
        @CallingScript = 'AnalyseMVUseMetadata',
        @CallingCode = 'Update query performance metadata',
        @LogMessage = @LogMessage

-- For each new_query_text in ##q_mv_link where the original costs have been obtained,
execute the new query
-- then using the subsequent plan handle, query the DB stats to get the cost, rows, reads
required and calculate columns.
DECLARE cur_ForEachNewQuery CURSOR LOCAL FAST_FORWARD FOR
        SELECT mv_link_id, new_query_text
        FROM ##q_mv_link link
        WHERE link.original_query_cost IS NOT NULL
        AND link.new_query_cost IS NULL
DECLARE @thisMVLinkID INT, @thisNewQueryText NVARCHAR(MAX), @thisPlanHandle VARBINARY(MAX)
DECLARE @d NVARCHAR(MAX)
OPEN cur_ForEachNewQuery
FETCH NEXT FROM cur_ForEachNewQuery INTO @thisMVLinkID, @thisNewQueryText
WHILE @@FETCH_STATUS = 0
BEGIN
        BEGIN TRY
                PRINT 'Attempting to execute new query, ##mv_link_id = ' +
CAST(@thisMVLinkID AS VARCHAR(10)) + '...'
                EXEC sp_executesql @thisNewQueryText
                PRINT @thisNewQueryText
                PRINT 'Executed query.'
                -- get plan handle from cache
                SET @d = @thisNewQueryText
                exec sp_executesql @d
                SELECT @thisPlanHandle = COALESCE(cp.plan_handle, cp.parent_plan_handle)
                FROM sys.dm_exec_cached_plans cp
                CROSS APPLY sys.dm_exec_sql_text(plan_handle) t
                CROSS APPLY sys.dm_exec_query_plan(plan_handle) q
                WHERE t.[text] = @thisNewQueryText

                IF @thisPlanHandle IS NOT NULL
                BEGIN
                        PRINT 'Found plan handle.'
                        DELETE FROM @results
                        ;WITH XMLNAMESPACES ( DEFAULT
N'http://schemas.microsoft.com/sqlserver/2004/07/showplan' ),
                        PlanText AS ( SELECT CAST(q.query_plan AS XML) AS QueryPlan,
                                        s.last_logical_reads,
                                        s.plan_handle
                                        FROM sys.dm_exec_query_stats s
                                        CROSS APPLY sys.dm_exec_text_query_plan
(s.plan_handle, s.statement_start_offset, s.statement_end_offset) q
                                        WHERE s.plan_handle = @thisPlanHandle
                                ),
                        PlanElements AS (
                                        SELECT PlanText.plan_handle,
                                                PlanText.QueryPlan,
                                                PlanText.last_logical_reads,

                                        RelOp.pln.value(N'@EstimatedTotalSubtreeCost', N'float') AS EstimatedCost,

                                        RelOp.pln.value(N'@EstimateRows', N'float') AS EstimateRows,
                                                RelOp.pln.value(N'@NodeId',
N'integer') AS NodeId

                                FROM PlanText

```

```

                                CROSS APPLY
PlanText.QueryPlan.nodes(N'//RelOp')RelOp(pln)
                                )

                                INSERT INTO @results ( plan_handle, last_logical_reads,
estimated_cost, estimated_rows )
                                SELECT e.plan_handle, e.last_logical_reads,
e.EstimatedCost, e.EstimateRows
                                FROM PlanElements e
                                WHERE e.NodeId = 0
                                IF @@ROWCOUNT > 0
                                PRINT 'Found query metadata.'
                                ELSE
                                PRINT 'Did NOT find query metadata.'
                                UPDATE link
                                SET
                                link.new_plan_handle = @thisPlanHandle,
                                link.new_query_cost = CAST(r.estimated_cost AS
NUMERIC(24,5)),
                                link.new_query_rows = r.estimated_rows,
                                link.new_query_read_count = r.last_logical_reads
                                FROM ##q_mv_link link
                                INNER JOIN @results r ON @thisPlanHandle = r.plan_handle
                                WHERE link.mv_link_id = @thisMVLinkID

                                PRINT 'Updated ##q_mv_link table with ' + CAST(@@ROWCOUNT AS VARCHAR(10)) +
' row.'
                                END
                                END TRY
                                BEGIN CATCH
                                PRINT ERROR_MESSAGE()
                                END CATCH
                                SET @thisPlanHandle = NULL
                                FETCH NEXT FROM cur_ForEachNewQuery INTO @thisMVLinkID, @thisNewQueryText
                                END
                                CLOSE cur_ForEachNewQuery
                                DEALLOCATE cur_ForEachNewQuery

-- Finally, calculate the query efficiencies and calculate the cost and efficiency deltas.
-- Efficiency as rows over reads as per document.
UPDATE link
SET
    link.original_query_efficiency =
        CAST((CAST(link.original_query_rows AS FLOAT) /
        CAST(CASE WHEN link.original_query_read_count = 0 THEN 1 ELSE
link.original_query_read_count END AS FLOAT))
        *100.0 AS NUMERIC(24,2)),
    link.new_query_efficiency =
        CAST((CAST(link.new_query_rows AS FLOAT) /
        CAST(CASE WHEN link.new_query_read_count = 0 THEN 1 ELSE
link.new_query_read_count END AS FLOAT))
        *100.0 AS NUMERIC(24,2))
FROM ##q_mv_link link
WHERE original_query_rows IS NOT NULL
AND original_query_read_count IS NOT NULL
AND new_query_rows IS NOT NULL
AND new_query_read_count IS NOT NULL

-- Cap off the efficiencies at 100% (for cases where fewer reads required than rows
returned).
UPDATE link
SET
    link.original_query_efficiency =
        CASE WHEN original_query_efficiency > 100 THEN 100.0 ELSE
original_query_efficiency END,
    link.new_query_efficiency =

```

```

CASE WHEN new_query_efficiency > 100 THEN 100.0 ELSE
new_query_efficiency END
FROM    ##q_mv_link link

UPDATE link
SET      link.cost_delta = new_query_cost - original_query_cost,
        link. efficiency_delta = new_query_efficiency - original_query_efficiency
FROM    ##q_mv_link link

-- Destroy any MVs extant in the DB that aren't listed in the ##mv table
DECLARE cur_ForEachView CURSOR LOCAL FAST_FORWARD FOR
        SELECT v.name
        FROM   tpcc.sys.views v
        LEFT  JOIN ##mv mv
        ON    v.[name] =
LTRIM(RTRIM(REPLACE(REPLACE(LEFT(mv.associated_view_definition, 50), 'CREATE VIEW [' , ''),
']', '')))
        WHERE LTRIM(RTRIM(REPLACE(REPLACE(LEFT(mv.associated_view_definition, 50),
'CREATE VIEW [' , ''), '] , ''))) IS NULL
DECLARE @thisView VARCHAR(255)
DECLARE @dSQL NVARCHAR(MAX)
SET @count = 0
OPEN   cur_ForEachView
FETCH NEXT FROM cur_ForEachView INTO @thisView
WHILE @@FETCH_STATUS = 0
BEGIN
        SET @dSQL = 'DROP VIEW [' + @thisView + ']'
        BEGIN TRY
                EXEC tpcc..sp_executesql @dSQL
                SET @count += 1
        END TRY
        BEGIN CATCH
        END CATCH
        FETCH NEXT FROM cur_ForEachView INTO @thisView
END
CLOSE cur_ForEachView
DEALLOCATE cur_ForEachView

SET @LogMessage = 'Dropped ' + CAST(@count AS VARCHAR(15)) + ' MVs that no longer exist in
##mv'
EXEC tpcc_queries.dbo.LogEntry
        @CallingScript = 'AnalyseMVUseMetadata',
        @CallingCode = 'Drop extant MVs',
        @LogMessage = @LogMessage

```

E.4 Implementation of the create and destroy MVs component

Algorithm:

| |
|---|
| Algorithm name: Analyse MV / Use Metadata |
| Inputs: RDBMS query plan cache, ##mv, ##mv_link, ##q global temporary tables |
| Outputs: Updated performance info. in ##mv_link, dropped MVs in DB |
| <pre>Declare a local temporary table @result with columns as follows: ---- Plan handle, ---- Last logical reads, ---- Estimated cost, ---- Estimated rows For each query present in ##mv/##mv_link joined on mv_id: ---- Delete contents of @results ---- *Parse query plan from query plan cache using XML document definition: ---- ---- Insert the following fields into @results from the output parse: ---- ---- ---- Plan handle, last logical reads, ---- ---- ---- Estimated cost, estimated rows. Update ##mv link with the data mapped as follows, keyed on plan_handle: ---- (@mv_link)Original query cost <- (@results)Estimated cost ---- (@mv_link)Original query read count <- (@results)Last logical reads ---- (@mv_link)Original query rows <- (@results)Estimated rows ---- Where: ---- ---- (@mv_link) Query cost is null OR ** (@mv_link) Query rows is null For each query in ##mv_link where costs were successfully obtained: ---- Execute new query ##mv_link.new_query_text ---- Fetch query execution statistics using process marked as * through to ** above. Calculate query statistic deltas (efficiency E): ---- For all rows in ##mv_link: ---- ---- Update original_query_efficiency: ---- ---- ---- Set to original_query_rows / (min: 1)(original_query_read_count) ---- ---- Update new_query_efficiency: ---- ---- ---- Set to new_query_rows / (min:1) (new_query_read_count) ---- ---- Where:All columns as above exist. For all rows in ##mv_link: ---- Where reads < rows returned (due to RDBMS efficiencies/caching): ---- ---- Set original new query efficiency = 1 ---- Set cost delta = new - original query cost ---- Set efficiency delta = new - original query efficiency Destroy any extant MVs: ---- For all MVs existing in the DB: ---- ---- Anti-join to ##mv table ---- ---- If not exists, drop MV</pre> |

Code Listing:

```
USE tpcc
SET NOCOUNT ON
DECLARE @dSQL NVARCHAR(MAX)
DECLARE @LogMessage VARCHAR(MAX)
```

```

-- Remove own queries
DELETE q
FROM   ##q q
WHERE  q.query_text LIKE ('##q%')
OR     q.query_text LIKE ('##mv%')
OR     q.query_text LIKE ('%g %')
OR     q.query_text LIKE ('%tpcc_queries%')

-- Remove queries with CROSS JOINS - hangs the process
DELETE q
FROM   ##q q
WHERE  q.query_text LIKE ('%CROSS%JOIN%')

SET @LogMessage = 'Deleted ' + CAST(@@ROWCOUNT AS VARCHAR(15)) + ' rows from ##q'
EXEC tpcc_queries.dbo.LogEntry
        @CallingScript = 'CreateAndDestroyMVs',
        @CallingCode = 'Remove own queries',
        @LogMessage = @LogMessage

-- Remove MVs no longer present in ##q together with their table entries
--SET @dSQL = N''
--DECLARE cur_ForEachMVToDelete CURSOR LOCAL FAST_FORWARD FOR
--      SELECT REPLACE(LEFT(mv.associated_view_definition, 50), 'CREATE', 'DROP')
--      FROM    ##mv mv
--      INNER  JOIN ##q_mv_link link ON mv.mv_id = link.mv_id
--      LEFT   JOIN ##q q ON link.plan_handle = q.plan_handle
--      WHERE  q.plan_handle IS NULL
DECLARE @count INT = 0
--OPEN cur_ForEachMVToDelete
--FETCH NEXT FROM cur_ForEachMVToDelete INTO @dSQL
--WHILE @@FETCH_STATUS = 0
--BEGIN
--      BEGIN TRY
--            SET @dSQL = REPLACE(REPLACE(@dSQL, CHAR(13), ''), CHAR(10), '')
--            PRINT @dSQL
--            PRINT '.'
--            EXEC tpcc..sp_executesql @dSQL
--            SELECT @@ROWCOUNT
--            SET @count += 1
--      END TRY
--      BEGIN CATCH
--      END CATCH
--      FETCH NEXT FROM cur_ForEachMVToDelete INTO @dSQL
--END
--CLOSE cur_ForEachMVToDelete
--DEALLOCATE cur_ForEachMVToDelete

--SET @LogMessage = 'Dropped ' + CAST(@count AS VARCHAR(15)) + ' MVs'
--EXEC tpcc_queries.dbo.LogEntry
--      @CallingScript = 'CreateAndDestroyMVs',
--      @CallingCode = 'Remove MVs no longer present in ##q',
--      @LogMessage = @LogMessage

--DELETE mv
--FROM ##mv mv
--INNER JOIN ##q_mv_link link ON mv.mv_id = link.mv_id
--LEFT JOIN ##q q ON link.plan_handle = q.plan_handle
--WHERE q.plan_handle IS NULL

--SET @LogMessage = 'Deleted ' + CAST(@@ROWCOUNT AS VARCHAR(15)) + ' entries from ##mv'
--EXEC tpcc_queries.dbo.LogEntry
--      @CallingScript = 'CreateAndDestroyMVs',
--      @CallingCode = 'Remove MVs no longer present in ##q',
--      @LogMessage = @LogMessage

```

```

--DELETE      link
--FROM      ##q_mv_link link
--LEFT JOIN ##mv mv ON link.mv_id = mv.mv_id
--WHERE mv.mv_id IS NULL
---- delete duplicates
--;WITH distincts AS (
--      SELECT MAX(mv_link_id) [max], plan_handle, mv_id
--      FROM ##q_mv_link
--      GROUP BY plan_handle, mv_id )
--DELETE      link
--FROM      ##q_mv_link link
--LEFT JOIN distincts d ON link.mv_link_id = d.[max]
--WHERE d.[max] IS NULL

--SET @LogMessage = 'Deleted ' + CAST(@@ROWCOUNT AS VARCHAR(15)) + ' entries from
##q_mv_link'
--EXEC tpcc_queries.dbo.LogEntry
--      @CallingScript = 'CreateAndDestroyMVs',
--      @CallingCode = 'Remove MVs no longer present in ##q',
--      @LogMessage = @LogMessage

-- Start query parsing
DECLARE cur_ForEachNinQ CURSOR LOCAL FAST_FORWARD FOR
-- Can modify with TOP N PERCENT
SELECT  plan_handle, usecounts, query_text, attributes, datasources, predicates
FROM    ##q q
WHERE   suitable_candidate IS NULL
ORDER  BY q.usecounts DESC
DECLARE @thisPlanHandle VARBINARY(8000), @thisUsecounts BIGINT, @thisQueryText
VARCHAR(MAX)
DECLARE @thisAttributes VARCHAR(MAX), @thisDatasources VARCHAR(MAX), @thisPredicates
VARCHAR(MAX)
DECLARE @c INT = 0
OPEN    cur_ForEachNinQ
FETCH  NEXT FROM cur_ForEachNinQ INTO @thisPlanHandle, @thisUseCounts, @thisQueryText,
@thisAttributes, @thisDatasources, @thisPredicates
WHILE  @@FETCH_STATUS = 0
BEGIN
-- Does query match standard SELECT, FROM, WHERE?
-- First check if there is nesting - remove these from scope
IF SUBSTRING(@thisAttributes, 7, 8000) LIKE ('%SELECT%')
OR SUBSTRING(@thisAttributes, 7, 8000) LIKE ('%FROM%')
OR SUBSTRING(@thisAttributes, 7, 8000) LIKE ('%WHERE%')
UPDATE ##q SET suitable_candidate = 0 WHERE plan_handle =
@thisPlanHandle
IF SUBSTRING(@thisDatasources, 5, 8000) LIKE ('%SELECT%')
OR SUBSTRING(@thisDatasources, 5, 8000) LIKE ('%FROM%')
OR SUBSTRING(@thisDatasources, 5, 8000) LIKE ('%WHERE%')
UPDATE ##q SET suitable_candidate = 0 WHERE plan_handle =
@thisPlanHandle
IF SUBSTRING(@thisPredicates, 6, 8000) LIKE ('%SELECT%')
OR SUBSTRING(@thisPredicates, 6, 8000) LIKE ('%FROM%')
OR SUBSTRING(@thisPredicates, 6, 8000) LIKE ('%WHERE%')
UPDATE ##q SET suitable_candidate = 0 WHERE plan_handle =
@thisPlanHandle
-- Now check if there is any use of system functions in the datasources - remove
from scope
IF @thisDatasources LIKE ('%(%)%')
UPDATE ##q SET suitable_candidate = 0 WHERE plan_handle =
@thisPlanHandle
-- Check that we have, at least, a FROM clause - exclude any queries with no
explicit datasources (like SELECT 1)
IF @thisDatasources IS NULL OR LEN(@thisDatasources) = 0

```

```

UPDATE ##q SET suitable_candidate = 0 WHERE plan_handle =
@thisPlanHandle
-- Do there exist identical queries with different predicates in ##q? If so group
them up with pipe delimitation.
SELECT @c = COUNT(*) FROM (
    SELECT q.attributes, q.datasources
    FROM ##q q
    WHERE q.attributes = @thisAttributes
    AND q.datasources = @thisDatasources
    AND ISNULL(q.predicates, '') != ISNULL(q.predicates, '') )
x
    IF @c > 1
    BEGIN
        UPDATE q
        SET q.grouped_predicates =
ISNULL(q.grouped_predicates, '') + '|' + q.predicates
        FROM ##q q
        WHERE plan_handle = @thisPlanHandle
    END
    -- If the query isn't already marked as unsuitable, mark as suitable
    IF ( SELECT TOP 1 suitable_candidate FROM ##q WHERE plan_handle =
@thisPlanHandle ) IS NULL
    BEGIN
        UPDATE q SET q.suitable_candidate = 1 FROM ##q q WHERE
q.plan_handle = @thisPlanHandle
    END
    FETCH NEXT FROM cur_ForEachNinQ INTO @thisPlanHandle, @thisUseCounts,
@thisQueryText, @thisAttributes, @thisDatasources, @thisPredicates
END
CLOSE cur_ForEachNinQ
DEALLOCATE cur_ForEachNinQ

-- Where the data sources are identical, group the predicates and apply to all
grouped_predicates
DECLARE cur_ForEachDistinctDatasource CURSOR LOCAL FAST_FORWARD FOR
SELECT DISTINCT q.datasources
FROM ##q q
WHERE q.predicates != ''
AND q.suitable_candidate = 1
AND q.grouped_predicates IS NULL
DECLARE @thisDS VARCHAR(MAX), @predicateList VARCHAR(MAX) = ''
OPEN cur_ForEachDistinctDatasource
FETCH NEXT FROM cur_ForEachDistinctDatasource INTO @thisDS
WHILE @@FETCH_STATUS = 0
BEGIN
    ;WITH predicates AS (
        SELECT DISTINCT q.predicates [p]
        FROM ##q q
        WHERE q.datasources = @thisDS
        AND q.predicates != ''
        AND q.suitable_candidate = 1
    )
    SELECT @predicateList =
CASE WHEN @predicateList = ''
THEN predicates.p
ELSE @predicateList + COALESCE(', ' + predicates.p, '')
END
FROM predicates
UPDATE q
SET q.grouped_predicates = @predicateList
FROM ##q q
WHERE q.datasources = @thisDS
AND q.predicates != ''
AND q.suitable_candidate = 1

```

```

        SET @predicateList = ''
        FETCH NEXT FROM cur_ForEachDistinctDatasource INTO @thisDS
    END
CLOSE cur_ForEachDistinctDatasource
DEALLOCATE cur_ForEachDistinctDatasource

-- Now remove all queries from system datasources from scope
DELETE q
FROM ##q q
WHERE q.datasources LIKE ('%master.%')
OR      q.datasources LIKE ('%model.%')
OR      q.datasources LIKE ('%msdb.%')
OR      q.datasources LIKE ('%tempdb.%')

SET @LogMessage = 'Deleted ' + CAST(@@ROWCOUNT AS VARCHAR(15)) + ' entries from ##q'
EXEC tpcc_queries.dbo.LogEntry
        @CallingScript = 'CreateAndDestroyMVs',
        @CallingCode = 'Remove all queries from system datasources from scope',
        @LogMessage = @LogMessage

DROP TABLE IF EXISTS #g
CREATE TABLE #g ( plan_handle VARBINARY(MAX), attributes VARCHAR(MAX),
original_attributes VARCHAR(MAX),
                    datasources VARCHAR(MAX), predicates VARCHAR(MAX),
original_predicates VARCHAR(MAX) )
INSERT INTO #g ( plan_handle, attributes, original_attributes, datasources, predicates,
original_predicates )
        SELECT q.plan_handle, q.attributes, q.attributes, q.datasources, q.predicates,
q.predicates
        FROM ##q q
        WHERE q.suitable_candidate = 1

SET @LogMessage = 'Inserted ' + CAST(@@ROWCOUNT AS VARCHAR(15)) + ' entries into #g'
EXEC tpcc_queries.dbo.LogEntry
        @CallingScript = 'CreateAndDestroyMVs',
        @CallingCode = 'Identify suitable queries from ##q into #g',
        @LogMessage = @LogMessage

-- b) group queries by same datasources
-- for each distinct datasource, aggregate the distinct attributes
DECLARE cur_ForEachDS CURSOR LOCAL FAST_FORWARD FOR
        SELECT DISTINCT g.datasources
        FROM #g g
SET @thisDS = ''
SET @thisAttributes = ''
DECLARE @thisAttributesSplit VARCHAR(MAX) = ''
OPEN cur_ForEachDS
FETCH NEXT FROM cur_ForEachDS INTO @thisDS
WHILE @@FETCH_STATUS = 0
BEGIN
        SELECT @thisAttributes = LEFT(x.attributes, LEN(x.attributes) - 1)
        FROM (
                SELECT g.attributes + ','
                FROM #g g
                WHERE g.datasources = @thisDS
                FOR XML PATH ('') ) x (attributes)

        -- This results in a comma-separated list of non-distinct attributes in
@thisAttributes for @thisDS
        -- Now de-duplicate this list
        SET @thisAttributesSplit = ''
        ;WITH splits AS (
                SELECT DISTINCT s.[value]

```

```

FROM string_split(@thisAttributes, ',') s )

SELECT @thisAttributesSplit = LEFT(x.attributes, LEN(x.attributes) - 1)
FROM (
    SELECT splits.[value] + ','
    FROM splits
    FOR XML PATH ('') ) x (attributes)

SET @thisAttributes = @thisAttributesSplit
-- Now update #g with the new deduplicated list of attributes
UPDATE g
SET g.attributes = 'SELECT ' + REPLACE(REPLACE(@thisAttributes,
'SELECT', ''), CHAR(8), ' ')
FROM #g g
WHERE g.datasources = @thisDS

SET @thisAttributes = ''
SET @thisAttributesSplit = ''
FETCH NEXT FROM cur_ForEachDS INTO @thisDS
END
CLOSE cur_ForEachDS
DEALLOCATE cur_ForEachDS

-- Now we need to alias each attribute to avoid column name collisions later when creating
MVs.
-- We string-split attributes by comma, ignoring the initial SELECT, append the alias as '
AS [alias]'
-- then glue everything back together again.
DECLARE cur_ForEachAttributes CURSOR LOCAL FAST_FORWARD FOR
    SELECT g.plan_handle, g.attributes
    FROM #g g
DECLARE @splits TABLE ( [value] VARCHAR(MAX) )
OPEN cur_ForEachAttributes
FETCH NEXT FROM cur_ForEachAttributes INTO @thisPlanHandle, @thisAttributes
WHILE @@FETCH_STATUS = 0
BEGIN
    INSERT INTO @splits
        SELECT DISTINCT value FROM string_split(@thisAttributes, ',')
    UPDATE @splits
    SET [value] = REPLACE([value], 'SELECT ', '')
    UPDATE @splits
    SET value = value + ' AS ' + '[' + REPLACE(REPLACE(REPLACE(value, '[' ,
'),', '' ), '' ), '' + ']'
    SELECT @thisAttributes = LEFT(x.attributes, LEN(x.attributes) - 1)
    FROM (
        SELECT s.value + ','
        FROM @splits s
        FOR XML PATH ('') ) x (attributes)
    SET @thisAttributes = 'SELECT ' + @thisAttributes
    UPDATE g
    SET g.attributes = @thisAttributes
    FROM #g g
    WHERE g.plan_handle = @thisPlanHandle

    DELETE FROM @splits
    FETCH NEXT FROM cur_ForEachAttributes INTO @thisPlanHandle, @thisAttributes
END
CLOSE cur_ForEachAttributes
DEALLOCATE cur_ForEachAttributes

-- Now we do it again for the original_attributes since the data source has changed.
DECLARE cur_ForEachAttributes CURSOR LOCAL FAST_FORWARD FOR
    SELECT g.plan_handle, g.original_attributes
    FROM #g g

```

```

DELETE FROM @splits
OPEN cur_ForEachAttributes
FETCH NEXT FROM cur_ForEachAttributes INTO @thisPlanHandle, @thisAttributes
WHILE @@FETCH_STATUS = 0
BEGIN
    INSERT INTO @splits
        SELECT DISTINCT value FROM string_split(@thisAttributes, ',')
    UPDATE @splits
    SET [value] = REPLACE([value], 'SELECT ', '')
    UPDATE @splits
    SET value = value + ' AS ' + '[' + REPLACE(REPLACE(REPLACE(value, '[',
'','],'), '), ') + ']'
    SELECT @thisAttributes = LEFT(x.attributes, LEN(x.attributes) - 1)
    FROM (
        SELECT s.value + ','
        FROM @splits s
        FOR XML PATH ('') ) x (attributes)
    SET @thisAttributes = 'SELECT ' + @thisAttributes
    UPDATE g
    SET g.original_attributes = @thisAttributes
    FROM #g g
    WHERE g.plan_handle = @thisPlanHandle

    DELETE FROM @splits
    FETCH NEXT FROM cur_ForEachAttributes INTO @thisPlanHandle, @thisAttributes
END
CLOSE cur_ForEachAttributes
DEALLOCATE cur_ForEachAttributes

-- schema binding fails if the datasources aren't in two-part names.
-- replace each datasource with dbo.<datasource> if it isn't already a two-part name i.e.
named schema.
-- address the simple case first, where FROM <word> exists, replace with FROM dbo.<word>
UPDATE g
SET g.datasources = REPLACE(g.datasources, 'FROM ', 'FROM dbo.')
FROM #g g
WHERE g.datasources NOT LIKE ('%.%') -- no dot therefore no joins

SET @LogMessage = 'Updated ' + CAST(@@ROWCOUNT AS VARCHAR(15)) + ' entries in #g with two-
part names'
EXEC tpcc_queries.dbo.LogEntry
    @CallingScript = 'CreateAndDestroyMVs',
    @CallingCode = 'Update #g with two-part names - simple case',
    @LogMessage = @LogMessage

-- Now address the complex case.
-- Where exists a space + datasource name, this must be a table.
-- If JOINS are involved we can replace the JOIN with a JOIN + ' ' + 'dbo.'
-- Won't work for views involving multiple schemas but this is rare and out of scope for
PoC.
UPDATE g
SET g.datasources = REPLACE(REPLACE(g.datasources, 'JOIN', 'JOIN dbo.'),
'FROM', 'FROM dbo.')
FROM #g g
WHERE g.datasources LIKE ('%JOIN%')

SET @LogMessage = 'Updated ' + CAST(@@ROWCOUNT AS VARCHAR(15)) + ' entries in #g with two-
part names'
EXEC tpcc_queries.dbo.LogEntry
    @CallingScript = 'CreateAndDestroyMVs',
    @CallingCode = 'Update #g with two-part names - JOIN case',
    @LogMessage = @LogMessage

```

```

UPDATE g
SET      g.datasources = REPLACE(g.datasources, 'dbo. ', 'dbo.')
FROM    #g g
WHERE   g.datasources LIKE ('%dbo. %')

SET @LogMessage = 'Updated ' + CAST(@@ROWCOUNT AS VARCHAR(15)) + ' entries in #g with two-
part names'
EXEC tpcc_queries.dbo.LogEntry
        @CallingScript = 'CreateAndDestroyMVs',
        @CallingCode = 'Update #g with two-part names - dbo. space case',
        @LogMessage = @LogMessage

-- for each distinct datasource, aggregate the predicates replacing all ANDs with ORs
-- as this will ensure 100% coverage of all predicates necessary for all queries for those
datasources
-- (performance issues could happen here)
-- store the original predicates in the g.original_predicates column for computation of
the new_query_text

DECLARE cur_ForEachDS CURSOR LOCAL FAST_FORWARD FOR
        SELECT DISTINCT g.datasources
        FROM    #g g
SET @thisDS = ''
SET @thisPredicates = ''
DECLARE @thisPredicatesSplit VARCHAR(MAX) = ''
OPEN    cur_ForEachDS
SET      @count = 0
FETCH   NEXT FROM cur_ForEachDS INTO @thisDS
WHILE   @@FETCH_STATUS = 0
BEGIN
        SELECT @thisPredicates = LEFT(x.predicates, LEN(x.predicates) - 1)
        FROM    (
                SELECT g.predicates + ','
                FROM    #g g
                WHERE   g.datasources = @thisDS
                FOR      XML PATH ('') ) x (predicates)

        -- This results in a comma-separated list of non-distinct predicates in
@thisPredicates for @thisDS
        -- Now de-duplicate this list
        SET @thisPredicatesSplit = ''
        ;WITH splits AS (
                SELECT DISTINCT s.[value]
                FROM    string_split(@thisPredicates, ',') s )

        SELECT @thisPredicatesSplit = LEFT(x.predicates, LEN(x.predicates) - 1)
        FROM    (
                SELECT splits.[value] + ','
                FROM    splits
                FOR      XML PATH ('') ) x (predicates)

        SET @thisPredicates = @thisPredicatesSplit

        -- Now replace commas with OR statements as these are predicates
        SET @thisPredicates = REPLACE(@thisPredicates, ',', ' OR ')

        -- Now update #g with the new deduplicated list of predicates
        UPDATE g
        SET      g.predicates = @thisPredicates
        FROM    #g g
        WHERE   g.datasources = @thisDS

        SET @count += 1

```

```

        SET @thisPredicates = ''
        SET @thisPredicatesSplit = ''
        FETCH NEXT FROM cur_ForEachDS INTO @thisDS
    END
    CLOSE cur_ForEachDS
    DEALLOCATE cur_ForEachDS

    SET @LogMessage = 'Updated ' + CAST(@count AS VARCHAR(15)) + ' entries in #g to
deduplicate predicates'
    EXEC tpcc_queries.dbo.LogEntry
        @CallingScript = 'CreateAndDestroyMVs',
        @CallingCode = 'Predicate deduplication',
        @LogMessage = @LogMessage

-- Fix the 'WHERE OR' / 'OR WHERE' issue
    UPDATE g
    SET g.predicates = REPLACE(g.predicates, 'WHERE OR', 'OR')
    FROM #g g
    UPDATE g
    SET g.predicates = REPLACE(g.predicates, 'OR WHERE', 'OR')
    FROM #g g
    UPDATE g
    SET g.predicates = 'WHERE ' + RIGHT(g.predicates, LEN(g.predicates) - 2)
    FROM #g g
    WHERE LEFT(LTRIM(g.predicates), 2) = 'OR'
    UPDATE g
    SET g.predicates = REPLACE(g.predicates, 'WHERE R', 'WHERE')
    FROM #g g

-- for each resulting distinct expression in #g, script it as an MV if it doesn't already
exist
    INSERT INTO ##mv (associated_view_definition)
        SELECT DISTINCT CAST('CREATE VIEW [<<NEWID>>] WITH SCHEMABINDING AS ' +
            g.attributes + ' ' + g.datasources + ' ' +
g.predicates AS VARCHAR(8000))
        FROM #g g
        LEFT JOIN ##mv mv ON (g.attributes + ' ' + g.datasources + ' ' + g.predicates) =
            REPLACE(RIGHT(mv.associated_view_definition, LEN(mv.associated_view_definition) -
73), 'ELECT', 'SELECT')
            WHERE REPLACE(RIGHT(mv.associated_view_definition,
LEN(mv.associated_view_definition) - 73), 'ELECT', 'SELECT') IS NULL
            AND ISNULL(mv.mv_implemented, 0) = 0

    DECLARE @inserted INT = @@ROWCOUNT

-- deduplicate ##mv based on associated view definition - should solve the problem
    DELETE mv
    FROM ##mv mv
    WHERE mv.mv_id NOT IN (
    SELECT MAX(mv.mv_id)
    FROM ##mv mv
    GROUP BY SUBSTRING(mv.associated_view_definition, CHARINDEX('SELECT',
mv.associated_view_definition, 1), 8000) )

    DECLARE @deleted INT = @@ROWCOUNT

    SET @LogMessage = 'Inserted ' + CAST(@inserted - @deleted AS VARCHAR(15)) + ' entries into
##mv'
    EXEC tpcc_queries.dbo.LogEntry
        @CallingScript = 'CreateAndDestroyMVs',

```

```

        @CallingCode = 'Create new entries in ##mv if do not already exist',
        @LogMessage = @LogMessage

-- now create the newids
UPDATE mv
SET
    mv.associated_view_definition = REPLACE(mv.associated_view_definition,
'<<NEWID>>', NEWID())
FROM    ##mv mv

SET @LogMessage = 'Updated ' + CAST(@@ROWCOUNT AS VARCHAR(15)) + ' entries in ##mv'
EXEC tpcc_queries.dbo.LogEntry
        @CallingScript = 'CreateAndDestroyMVs',
        @CallingCode = 'Update the MV entries with new NEWID()s',
        @LogMessage = @LogMessage

-- fix a codepage issue
UPDATE mv
SET
    mv.associated_view_definition =
REPLACE(REPLACE(associated_view_definition, '&lt;', '<'), '&gt;', '>',
'>')
FROM    ##mv mv

UPDATE g
SET
    g.predicates =
REPLACE(REPLACE(predicates, '&lt;', '<'), '&gt;', '>', '>')
FROM    #g g

-- create the link entries, if they don't already exist
INSERT INTO ##q_mv_link ( mv_id, plan_handle, new_query_text, date_created )
    SELECT mv.mv_id, g.plan_handle,
           g.original_attributes + ' ' +
           --g.attributes + ' ' +
           'FROM ' +
           SUBSTRING(mv.associated_view_definition, 13, 37) + ']' +
           --g.predicates [new_query_text],
           g.original_predicates [new_query_text], -- trial to test
performance improvement.
           GETDATE()
    FROM    ##mv mv
    INNER JOIN #g g
    ON      LTRIM(RTRIM(mv.attributes_datasources_predicates)) =
           LTRIM(RTRIM(g.attributes + ' ' + g.datasources + ' ' + g.predicates))

SET @LogMessage = 'Inserted ' + CAST(@@ROWCOUNT AS VARCHAR(15)) + ' entries into
##q_mv_link'
EXEC tpcc_queries.dbo.LogEntry
        @CallingScript = 'CreateAndDestroyMVs',
        @CallingCode = 'Create new entries in ##q_mv_link if do not already exist',
        @LogMessage = @LogMessage

-- clean up MV definition for double dbo issue
UPDATE mv
SET
    mv.associated_view_definition = REPLACE(mv.associated_view_definition,
'dbo.dbo.', 'dbo.')
FROM    ##mv mv
-- clean up MV definition for OR/der issue
UPDATE mv
SET
    mv.associated_view_definition = REPLACE(mv.associated_view_definition, '
der', 'order')
FROM    ##mv mv

-- Update the original query columns in ##q_mv_link
UPDATE link

```

```

SET          link.original_query_columns =
             LEN(SUBSTRING(q.query_text, 1, CHARINDEX('FROM', q.query_text, 1))) -
             LEN(REPLACE(SUBSTRING(q.query_text, 1, CHARINDEX('FROM',
q.query_text, 1)), ',', '')) + 1
FROM        ##q_mv_link link
INNER      JOIN ##q q ON link.plan_handle = q.plan_handle

-- for each MV, check MV parses.  Delete those that don't from both the link and mv
tables.
-- added top 1m clause to prevent performance hangs.
DECLARE cur_ForEachMV CURSOR LOCAL FAST_FORWARD FOR
          SELECT mv_id, mv.associated_view_definition
          FROM    ##mv mv
          WHERE   mv.mv_implemented IS NULL OR mv.mv_implemented = 0
DECLARE @thisMVID INT
DECLARE @thisView VARCHAR(MAX)
DECLARE @success BIT = 0
SET      @count = 0
DECLARE @failedCount INT = 0
DECLARE @rcounts TABLE ( r BIGINT )
OPEN     cur_ForEachMV
FETCH   NEXT FROM cur_ForEachMV INTO @thisMVID, @thisView
WHILE   @@FETCH_STATUS = 0
BEGIN
    SET @dSQL = CAST(@thisView AS NVARCHAR(MAX))
    BEGIN TRY
        -- Bug fix: Remove accidental CROSS JOIN conditions.
        IF @dSQL LIKE ('%orders.o_w_id = customer.c_w_id%')
        OR @dSQL LIKE ('%history.h_c_w_id = customer.c_w_id%')
        OR @dSQL LIKE ('%new_order.no_w_id = orders.o_w_id%')
        BEGIN
            SET @success = 0
        END
        ELSE BEGIN
            -- First check the expected row count.  If effectively a cross
join, abort.
            SET @dSQL = 'SELECT COUNT(*) FROM ( ' + SUBSTRING(@dSQL, 74, 8000)
+ ') X;'
            PRINT @dSQL
            INSERT INTO @rcounts
                EXEC tpcc..sp_executesql @dSQL
            IF ( SELECT TOP 1 r FROM @rcounts ) <= 1000000
                BEGIN
                    PRINT 'View passed row check test.  Proceeding to
create view...'
                    SET @dSQL = CAST(@thisView AS NVARCHAR(MAX))
                    PRINT @dSQL
                    EXEC tpcc..sp_executesql @dSQL
                    SET @success = 1
                    SET @count += 1
                END
            ELSE BEGIN
                SET @success = 0
            END
        END
    END TRY
    BEGIN CATCH
        PRINT ERROR_MESSAGE()
        PRINT 'Failed to create MV.'
        SET @success = 0
    END CATCH
    IF @success = 1
    BEGIN
        PRINT 'Successfully created MV'
    END

```

```

        UPDATE mv
        SET          mv.mv_implemented = 1
        FROM        ##mv mv
        WHERE       mv.mv_id = @thisMVID

    END
    ELSE
    BEGIN
        SET @failedCount += 1
        DELETE FROM ##q_mv_link WHERE mv_id = @thisMVID
        DELETE FROM ##mv WHERE mv_id = @thisMVID

    END
    SET @success = 0
    DELETE FROM @rcounts
    FETCH NEXT FROM cur_ForEachMV INTO @thisMVID, @thisView
END
CLOSE cur_ForEachMV
DEALLOCATE cur_ForEachMV

SET @LogMessage = 'Created ' + CAST(@count AS VARCHAR(15)) + ' new MVs'
EXEC tpcc_queries.dbo.LogEntry
    @CallingScript = 'CreateAndDestroyMVs',
    @CallingCode = 'Create new MVs',
    @LogMessage = @LogMessage

SET @LogMessage = 'Failed to create ' + CAST(@failedCount AS VARCHAR(15)) + ' new MVs'
EXEC tpcc_queries.dbo.LogEntry
    @CallingScript = 'CreateAndDestroyMVs',
    @CallingCode = 'Create new entries in ##mv if do not already exist',
    @LogMessage = @LogMessage

-- Create the clustered indexes on all the views - this materialises them away from the
base tables.
-- Try the first column only.
SET @count = 0
SET @failedCount = 0
DECLARE cur_ForEachDistinctView CURSOR LOCAL FAST_FORWARD FOR
    SELECT DISTINCT v.name [viewname], mv.mv_id
    FROM tpcc.sys.views v
    -- Exclude those views already considered.
    INNER JOIN ##mv mv ON mv.associated_view_definition LIKE ('%' + v.[name] + '%')
    WHERE mv.has_indexed_view IS NULL
DECLARE @thisViewName VARCHAR(255) = ''
DECLARE @cName VARCHAR(255)
OPEN cur_ForEachDistinctView
FETCH NEXT FROM cur_ForEachDistinctView INTO @thisViewName, @thisMVID
WHILE @@FETCH_STATUS = 0
BEGIN
    SET @cName = ( SELECT TOP 1 c.name
                  FROM tpcc.sys.columns c
                  INNER JOIN tpcc.sys.views v
                  ON c.object_id = v.object_id
                  WHERE column_id = 1
                  AND v.[name] = @thisViewName )
    SET @dSQL = 'CREATE UNIQUE CLUSTERED INDEX rid_' + REPLACE(@thisViewName, '-', '')
+
    ' ON dbo.[ ' + @thisViewName + ' ] ( [ ' + @cName + ' ] );'
    BEGIN TRY
        -- This will only succeed for views without dup keys, without LEFT/RIGHT
joins.
        -- The unindexed views will remain though, can address with other
strategies i.e. NCIXs.
        EXEC sp_executesql @dSQL
        UPDATE mv

```

```

        SET          mv.has_indexed_view = 1
    FROM      ##mv mv
    WHERE     mv.associated_view_definition LIKE ('%' + @thisViewName + '%')
    SET @count += 1
END TRY
BEGIN CATCH
    PRINT ERROR_MESSAGE()
    PRINT 'Attempting to replace the view with a table...'
    BEGIN TRY
        SET @dSQL = 'DROP VIEW [' + @thisViewName + '];'
        EXEC tpcc..sp_executesql @dSQL
        SET @dSQL = ( SELECT TOP 1 associated_view_definition FROM ##mv
WHERE mv_id = @thisMVID )
        SET @dSQL = REPLACE(@dSQL, 'CREATE VIEW ', 'SELECT * INTO ')
        SET @dSQL = REPLACE(@dSQL, 'WITH SCHEMABINDING AS', 'FROM (')
        SET @dSQL = @dSQL + ') X'
        PRINT @dSQL
        EXEC sp_executesql @dSQL
    END TRY
    BEGIN CATCH
        PRINT ERROR_MESSAGE()
        PRINT 'Table creation failed.'
        SET @failedCount += 1
        UPDATE mv
        SET          mv.has_indexed_view = 0
        FROM      ##mv mv
        WHERE     mv.associated_view_definition LIKE ('%' + @thisViewName +
%'')
    END CATCH
END CATCH
FETCH NEXT FROM cur_ForEachDistinctView INTO @thisViewName, @thisMVID
END
CLOSE cur_ForEachDistinctView
DEALLOCATE cur_ForEachDistinctView

SET @LogMessage = 'Created ' + CAST(@count AS VARCHAR(15)) + ' new indexes on MVs'
EXEC tpcc_queries.dbo.LogEntry
    @CallingScript = 'CreateAndDestroyMVs',
    @CallingCode = 'Create new unique clustered indexes on MVs',
    @LogMessage = @LogMessage

SET @LogMessage = 'Failed to create ' + CAST(@failedCount AS VARCHAR(15)) + ' new indexes
on MVs'
EXEC tpcc_queries.dbo.LogEntry
    @CallingScript = 'CreateAndDestroyMVs',
    @CallingCode = 'Create new unique clustered indexes on MVs',
    @LogMessage = @LogMessage

-- For those that failed i.e. queries aren't suitable for indexed views, we can use
database tables in the same way.
-- Duplicates are allowed on those and we may still see better performance.
-- First drop the view, then recreate it as a table (SELECT TOP 0 * FROM <view> INTO
<table_renamed>, drop view, rename table)
-- The new_query_text then remains valid.
-- Consider how you will drop tables rather than views.

-- YOU ARE HERE

-- For queries in ##q_mv_link, update the new_query_text to use [alias] square brackets
for the predicates
-- else it won't parse as the datasource has changed to the MV.

```

```

DECLARE cur_ForEachNewQuery CURSOR LOCAL FAST_FORWARD FOR
    SELECT link.mv_link_id, link.new_query_text
    FROM    ##q_mv_link link
OPEN      cur_ForEachNewQuery
FETCH    NEXT FROM cur_ForEachNewQuery INTO @thisMVID, @thisQueryText
WHILE    @@FETCH_STATUS = 0
BEGIN
    DELETE FROM @splits
    INSERT INTO @splits
        SELECT [value] FROM string_split(@thisQueryText, ' ')
    UPDATE @splits SET [value] = '[' + [value] + ']' WHERE [value] LIKE ('%.%')
    SELECT @thisQueryText = x.querytext -- LEFT(x.querytext, LEN(x.querytext) - 1)
    FROM    (
        SELECT s.value + ' '
        FROM    @splits s
        FOR      XML PATH ('') ) x (querytext)
    UPDATE link
    SET      link.new_query_text = @thisQueryText
    FROM    ##q_mv_link link
    WHERE    link.mv_link_id = @thisMVID
    FETCH    NEXT FROM cur_ForEachNewQuery INTO @thisMVID, @thisQueryText
END
CLOSE    cur_ForEachNewQuery
DEALLOCATE    cur_ForEachNewQuery

-- Fix XML codepage issues in new_query_text
UPDATE link
SET      link.new_query_text = REPLACE(REPLACE(REPLACE(link.new_query_text, '[' ,
    '['), ']' , ']' ), '&#x20;' , ' ')
FROM    ##q_mv_link link
UPDATE link
SET      link.new_query_text = REPLACE(link.new_query_text, ',]' , ', ')
FROM    ##q_mv_link link
UPDATE link
SET      link.new_query_text =
    REPLACE(REPLACE(REPLACE(REPLACE(link.new_query_text , '&lt;' , '<'),
    '&gt;' , '>'), '&lt;' , '<'), '&gt;' , '>')
FROM    ##q_mv_link link
UPDATE link
SET      link.new_query_text = REPLACE(REPLACE(link.new_query_text, 'WHERE' , ' WHERE
    '), 'FROM' , ' FROM ')
FROM    ##q_mv_link link
-- the following addresses a bug with square brackets - a workaround, to be fixed
(string_split probably at fault).
UPDATE link
SET      link.new_query_text = REPLACE(link.new_query_text, ']' , ']' , [w'])
FROM    ##q_mv_link link
UPDATE link
SET      link.new_query_text = REPLACE(link.new_query_text, ']' , ']' , [d'])
FROM    ##q_mv_link link
UPDATE link
SET      link.new_query_text = REPLACE(link.new_query_text, ']' , ']' , [c'])
FROM    ##q_mv_link link
UPDATE link
SET      link.new_query_text = REPLACE(link.new_query_text, ']' , ']' , [h'])
FROM    ##q_mv_link link
UPDATE link
SET      link.new_query_text = REPLACE(link.new_query_text, ']' , ']' , [i'])
FROM    ##q_mv_link link
UPDATE link
SET      link.new_query_text = REPLACE(link.new_query_text, ']' , ']' , [n'])
FROM    ##q_mv_link link
UPDATE link
SET      link.new_query_text = REPLACE(link.new_query_text, ']' , ']' , [o'])

```

```

FROM    ##q_mv_link link
UPDATE  link
SET      link.new_query_text = REPLACE(link.new_query_text, '],s', '], [s')
FROM    ##q_mv_link link

/*
(runs as async process populates efficiency data)
GROUP M.V.s AGAINST SUM OF EFFICIENCY DELTAS (WHERE EXIST) IN #MV
WHERE SUM(EFFICIENCY DELTA) <=0, DROP M.V.
*/
-- TO ADD WHEN Analyse SCRIPT IS RUNNING

--;WITH inefficientMVs AS (
--SELECT      link.mv_id, SUM(link.encyency_delta) [overall_efficiency_delta]
--FROM    ##q_mv_link link
--WHERE link.encyency_delta IS NOT NULL
--GROUP BY link.mv_id )
--DELETE      link
--FROM    ##q_mv_link link
--INNER JOIN inefficientMVs ON link.mv_id = inefficientMVs.mv_id
--WHERE overall_efficiency_delta < 0
--DELETE      mv
--FROM    ##mv mv
--LEFT JOIN ##q_mv_link link ON mv.mv_id = link.mv_id
--WHERE link.mv_id IS NULL

---- SUMMARY FOR TESTING
--SELECT      'Queries in cache: ', COUNT(*) [count]
--FROM    sys.dm_exec_cached_plans UNION ALL
--SELECT      'Queries in ##q: ', COUNT(*)
--FROM    ##q UNION ALL
--SELECT      'Queries in #g: ', COUNT(*)
--FROM    #g UNION ALL
--SELECT      'New MVs in ##mv: ', COUNT(*)
--FROM    ##mv UNION ALL
--SELECT      'Links in ##q_mv_link: ', COUNT(*)
--FROM    ##q_mv_link

--SELECT      q.query_text, mv.associated_view_definition, link.new_query_text
--FROM    ##q q
--INNER JOIN ##q_mv_link link ON q.plan_handle = link.plan_handle
--INNER JOIN ##mv mv ON link.mv_id = mv.mv_id

--TRUNCATE TABLE ##q_mv_link
--TRUNCATE TABLE ##mv
--TRUNCATE TABLE #g

```

E.5 Implementation of the Query Generator and Caller (for Testing Purposes)

```

WHILE 1=1
BEGIN

-- INIT
DROP TABLE IF EXISTS #t
DROP TABLE IF EXISTS #c
DROP TABLE IF EXISTS #w
DROP TABLE IF EXISTS #p
DROP TABLE IF EXISTS #s
CREATE TABLE #t ( tbl VARCHAR(255) )
CREATE TABLE #c ( col VARCHAR(255) )
CREATE TABLE #w ( col VARCHAR(255), y VARCHAR(255) )
DECLARE @thisWCol VARCHAR(255), @thisWY VARCHAR(255)
CREATE TABLE #p ( id TINYINT, primitive VARCHAR(15) )
CREATE TABLE #s ( id INT IDENTITY(1,1) PRIMARY KEY NOT NULL, s VARCHAR(MAX), f
VARCHAR(MAX), w VARCHAR(MAX) )
INSERT INTO #p
    SELECT 1, '=' UNION ALL
    SELECT 2, '<' UNION ALL
    SELECT 3, '>' UNION ALL
    SELECT 4, '<=' UNION ALL
    SELECT 5, '>=' UNION ALL
    SELECT 6, '!=' UNION ALL
    SELECT 7, 'IS NULL' UNION ALL
    SELECT 8, 'IS NOT NULL'
DECLARE @dSQL NVARCHAR(MAX)
DECLARE @SelectStmt VARCHAR(MAX) = '', @FromStmt VARCHAR(MAX) = '', @WhereStmt
VARCHAR(MAX) = ''
DECLARE @randomPrim VARCHAR(15)
-- Get random number of user tables > 1 from tpcc in any order
SET @dSQL = N'
INSERT INTO #t
    SELECT TOP ' + CAST(ABS(CHECKSUM(NEWID()))) % 8 + 2 AS CHAR(1)) +
    ' name FROM tpcc.sys.tables t WHERE t.type_desc = 'USER_TABLE' ORDER BY
NEWID();'
EXEC sp_executesql @dSQL
-- for each table, fetch a random % of the columns into list #c
DECLARE cur_forEachTable CURSOR FAST_FORWARD FOR
    SELECT tbl FROM #t
DECLARE @thisT VARCHAR(255)
OPEN cur_forEachTable
FETCH NEXT FROM cur_forEachTable INTO @thisT
WHILE @@FETCH_STATUS = 0
BEGIN
    SET @dSQL = N'
    INSERT INTO #c (col)
        SELECT TOP ' + CAST(ABS(CHECKSUM(NEWID()))) % 100 + 1 AS CHAR(3)) + '
PERCENT c.name
        FROM tpcc.sys.columns c INNER JOIN tpcc.sys.tables t ON c.object_id =
t.object_id
        WHERE t.name = ''' + @thisT + ''''
    EXEC sp_executesql @dSQL
    -- Convert column to comma-separated list
    -- Construct the SELECT statement for this table
    SELECT @SelectStmt = CASE
                                                                    WHEN @SelectStmt = ''
                                                                    THEN c.col
                                                                    ELSE @SelectStmt +
COALESCE(',' + c.col, '')

```

```

END
FROM #c c

-- Construct the FROM statement
SET @FromStmt = @thisT

-- Construct the WHERE statement
-- Get column datatypes
SET @dSQL = N'
INSERT INTO #w (col)
SELECT TOP ' + CAST(ABS(CHECKSUM(NEWID())) % 100 + 1 AS CHAR(3)) + ' PERCENT
c.name
FROM tpcc.sys.columns c INNER JOIN tpcc.sys.tables t ON c.object_id =
t.object_id
WHERE t.name = ''' + @thisT + '''
EXEC sp_executesql @dSQL
IF EXISTS ( SELECT * FROM #w )
UPDATE w
SET w.y = y.[name]
FROM tpcc.sys.columns c
INNER JOIN tpcc.sys.types y ON c.system_type_id = y.system_type_id
INNER JOIN #w w ON c.[name] = w.col
INNER JOIN tpcc.sys.tables t ON c.object_id = t.object_id
AND t.[name] = @thisT

-- Constrict predicates to numerics, ints, bits
DELETE FROM #w WHERE y NOT IN ('bigint','int','decimal','numeric','float','bit')

-- Fully-qualify the predicates (not using aliases)
UPDATE w SET w.col = @FromStmt + '.' + w.col FROM #w w

IF EXISTS ( SELECT * FROM #w )
BEGIN
DECLARE cur_ForEachWhere CURSOR LOCAL FAST_FORWARD FOR
SELECT w.col, w.y
FROM #w w
OPEN cur_ForEachWhere
FETCH NEXT FROM cur_ForEachWhere INTO @thisWCol, @thisWY
WHILE @@FETCH_STATUS = 0
BEGIN
IF @WhereStmt = ''
SET @WhereStmt = ''
-- Get a random primitive
SET @randomPrim = ( SELECT TOP 1 p.primitive FROM #p p ORDER BY
NEWID() )
IF @randomPrim = 'IS NULL' OR @randomPrim = 'IS NOT NULL'
SET @WhereStmt = @WhereStmt + @thisWCol + ' ' + @randomPrim
ELSE BEGIN
IF @thisWY = 'bit'
SET @WhereStmt = @WhereStmt + @thisWCol + ' ' +
@randomPrim + ' ' + CAST(ABS(CHECKSUM(NEWID())) % 2 AS CHAR(1))
IF @thisWY IN ('bigint', 'int')
SET @WhereStmt = @WhereStmt + @thisWCol + ' ' +
@randomPrim + ' ' + CAST(ABS(CHECKSUM(NEWID())) % 10000 AS VARCHAR(30))
IF @thisWY IN ('numeric','decimal')
SET @WhereStmt = @WhereStmt + @thisWCol + ' ' +
@randomPrim + ' ' +
CAST(ABS(CHECKSUM(NEWID())) % 10000 + 1 /
(ABS(CHECKSUM(NEWID())) % 10000 + 1) AS VARCHAR(30))
END
-- Random AND/OR selection
SET @WhereStmt = @WhereStmt +
CASE WHEN ABS(CHECKSUM(NEWID())) % 2 + 1 = 1 THEN ' AND '
ELSE ' OR ' END

```

```

        FETCH NEXT FROM cur_ForEachWhere INTO @thisWCol, @thisWY
    END
    CLOSE cur_ForEachWhere
    DEALLOCATE cur_ForEachWhere

    TRUNCATE TABLE #c
    TRUNCATE TABLE #w

    -- Fully-qualify the attributes (not using aliases)
    SET @SelectStmt = @FromStmt + '.' + REPLACE(@SelectStmt, ',', ', ' + @FromStmt +
'.')

    -- Trim the trailing AND/OR from the predicates
    SET @WhereStmt = CASE WHEN RIGHT(@WhereStmt, 4) = 'AND '
        THEN LEFT(@WhereStmt, LEN(@WhereStmt)
- 4)
        WHEN RIGHT(@WhereStmt, 4) = 'OR'
- 3)
        THEN LEFT(@WhereStmt, LEN(@WhereStmt)
        END

    -- Store the statements for later use
    INSERT INTO #s ( s, f, w)
        SELECT @SelectStmt, @FromStmt, @WhereStmt

    SET @SelectStmt = ''
    SET @FromStmt = ''
    SET @WhereStmt = ''
    END
    FETCH NEXT FROM cur_forEachTable INTO @thisT
END
CLOSE cur_forEachTable
DEALLOCATE cur_forEachTable

-- Process above yields table #s with select, from, where clauses.
-- Now decide on a number of joins to use, from 0-5.
-- Uses the relationships specified by the TPC-C benchmark dataset documentation.

UPDATE s SET s.w = '' FROM #s s WHERE s.w IS NULL

DROP TABLE IF EXISTS #j
CREATE TABLE #j (id TINYINT, jointype VARCHAR(20))
DECLARE @joinChance NUMERIC(5,2)
INSERT INTO #j (id, jointype)
    -- Add entries into this table by frequency, which we'll use as weighting
    -- Cross joins are relatively rare so only 1/100 chance, others accordingly.
    -- No joins are ''
    SELECT TOP 33 ROW_NUMBER() OVER ( ORDER BY ( SELECT NULL ) ) [id], 'INNER JOIN'
[jointype]
    FROM sys.objects
    UNION ALL
    SELECT TOP 33 ROW_NUMBER() OVER ( ORDER BY ( SELECT NULL ) ) + 33 [id], 'LEFT
JOIN' [jointype]
    FROM sys.objects
    UNION ALL
    SELECT TOP 33 ROW_NUMBER() OVER ( ORDER BY ( SELECT NULL ) ) + 66 [id], 'RIGHT
JOIN' [jointype]
    FROM sys.objects
    UNION ALL
    SELECT 100, 'CROSS JOIN'

-- Pick a JOIN to use. No join is a natural probability of no relationship existing.
DECLARE @thisJoin VARCHAR(20) = 'NONE'
SET @joinChance = ( SELECT ABS(CHECKSUM(NEWID())) % 100 + 1 )

```

```

        IF @joinChance <= 100
            SET @thisJoin = ( SELECT jointype FROM #j WHERE id = @joinChance )

-- Simple case - no JOINS.  Select a random entry from #s.
IF @thisJoin = 'NONE'
BEGIN
    SELECT TOP 1  @SelectStmt = 'SELECT ' + s.s,
                    @FromStmt = 'FROM ' + s.f,
                    @WhereStmt = CASE WHEN s.w != '' THEN 'WHERE ' +
s.w ELSE '' END
    FROM    #s s
    ORDER  BY NEWID()
END

-- Construct a relationship table to describe the constraints, based on the TPC-C
documentation
-- Cannot use system views as HammerDB does not formalise the relationships
-- @s1 = child, @s2 = parent
DROP TABLE IF EXISTS #r
CREATE TABLE #r ( id INT IDENTITY(1,1) NOT NULL PRIMARY KEY,
    s1 VARCHAR(255), s1c VARCHAR(255), s2 VARCHAR(255), s2c VARCHAR(255) )
INSERT INTO #r
    SELECT 'district', 'd_w_id', 'warehouse', 'w_id' UNION ALL
    SELECT 'customer', 'c_w_id', 'district', 'd_w_id' UNION ALL
    SELECT 'customer', 'c_d_id', 'district', 'd_id' UNION ALL
    SELECT 'history', 'h_c_w_id', 'customer', 'c_w_id' UNION ALL
    SELECT 'history', 'h_c_d_id', 'customer', 'c_d_id' UNION ALL
    SELECT 'history', 'h_c_id', 'customer', 'c_id' UNION ALL
    SELECT 'history', 'h_w_id', 'district', 'd_w_id' UNION ALL
    SELECT 'history', 'h_d_id', 'district', 'd_id' UNION ALL
    SELECT 'new_order', 'no_w_id', 'orders', 'o_w_id' UNION ALL
    SELECT 'new_order', 'no_d_id', 'orders', 'o_d_id' UNION ALL
    SELECT 'new_order', 'no_o_id', 'orders', 'o_id' UNION ALL
    SELECT 'orders', 'o_w_id', 'customer', 'c_w_id' UNION ALL
    SELECT 'orders', 'o_d_id', 'customer', 'c_d_id' UNION ALL
    SELECT 'orders', 'o_c_id', 'customer', 'c_id' UNION ALL
    SELECT 'order_line', 'ol_w_id', 'orders', 'o_w_id' UNION ALL
    SELECT 'order_line', 'ol_d_id', 'orders', 'o_d_id' UNION ALL
    SELECT 'order_line', 'ol_o_id', 'orders', 'o_id' UNION ALL
    SELECT 'order_line', 'ol_supply_w_id', 'stock', 's_w_id' UNION ALL
    SELECT 'order_line', 'ol_i_id', 'stock', 's_i_id' UNION ALL
    SELECT 'stock', 's_w_id', 'warehouse', 'w_id' UNION ALL
    SELECT 'stock', 's_i_id', 'item', 'i_id'

-- Inner, left, right JOIN
DECLARE @s1 INT, @s2 INT, @rN INT = 0
SELECT @s1 = ( SELECT TOP 1 s.id FROM #s s ORDER BY NEWID() )
SELECT @s2 = ( SELECT TOP 1 s.id FROM #s s WHERE s.id != @s1 ORDER BY NEWID() )
DECLARE @f1 VARCHAR(255), @f2 VARCHAR(255)
SELECT @f1 = ( SELECT s.f FROM #s s WHERE s.id = @s1 )
SELECT @f2 = ( SELECT s.f FROM #s s WHERE s.id = @s2 )
IF @thisJoin IN ('LEFT JOIN', 'RIGHT JOIN', 'INNER JOIN', 'CROSS JOIN')
BEGIN
    IF EXISTS ( SELECT r.id FROM #r r WHERE (r.s1 = @f1 AND r.s2 = @f2) OR (r.s2 = @f1
AND r.s1 = @f2) )
        BEGIN
            SET @rN = ( SELECT TOP 1 r.id FROM #r r
                WHERE (r.s1 = @f1 AND r.s2 = @f2) OR (r.s2 = @f1 AND
r.s1 = @f2) ORDER BY NEWID() )
            IF @thisJoin != 'CROSS JOIN'
                SELECT @FromStmt = r.s1 + ' ' + @thisJoin + ' ' + r.s2 + ' ON ' +
r.s1 + '.' + r.s1c + ' = ' + r.s2 + '.' + r.s2c
                FROM    #r r
                WHERE  r.id = @rN

```

```

        IF @thisJoin = 'CROSS JOIN'
            SELECT @FromStmt = r.s1 + ' ' + @thisJoin + ' ' + r.s2
            FROM #r r
            WHERE r.id = @rN
        SET @SelectStmt = ( SELECT s.s FROM #s s WHERE s.id = @s1 )
        SET @SelectStmt = @SelectStmt + ', '
        SET @SelectStmt = @SelectStmt + ( SELECT s.s FROM #s s WHERE s.id = @s2 )
        SET @WhereStmt = ( SELECT s.w FROM #s s WHERE s.id = @s1 )
        SET @WhereStmt = @WhereStmt + ' AND '
        SET @WhereStmt = @WhereStmt + ( SELECT s.w FROM #s s WHERE s.id = @s2 )
        IF LEFT(@WhereStmt, 5) = ' AND '
            SET @WhereStmt = RIGHT(@WhereStmt, LEN(@WhereStmt) - 4)
        IF RIGHT(@WhereStmt, 5) = ' AND '
            SET @WhereStmt = LEFT(@WhereStmt, LEN(@WhereStmt) - 4)
        SET @WhereStmt = LTRIM(RTRIM(@WhereStmt))
    END
ELSE BEGIN
    -- No relationship found
    SET @SelectStmt = ( SELECT TOP 1 s.s FROM #s s )
    SET @FromStmt = ( SELECT TOP 1 s.f FROM #s s )
    SET @WhereStmt = ( SELECT TOP 1 s.w FROM #s s )
END
END

SET @SelectStmt = 'SELECT ' + @SelectStmt
SET @FromStmt = 'FROM ' + @FromStmt
IF LEN(@WhereStmt) > 0
    SET @WhereStmt = 'WHERE ' + @WhereStmt

--PRINT @SelectStmt
--PRINT @FromStmt
--PRINT @WhereStmt

INSERT INTO tpcc_queries.dbo.queries ( query )
    SELECT @SelectStmt + ' ' + @FromStmt + ' ' + @WhereStmt

-- SELECT * FROM tpcc_queries.dbo.queries

END

```

```

import pyodbc, random, time, datetime

successCount = 0
failedCount = 0

def getConnection(db):
    conn = pyodbc.connect(
        "Driver={SQL Server Native Client 11.0};"
        "Server=localhost;"
        "Database=" + db + ";"
        "Trusted_Connection=yes;")
    conn.timeout = 20 # added to kill long-running queries
    return conn

def getRandomQuery(conn):
    r = random.randint(1, 10000)
    stmt = "SELECT query FROM tpcc_queries.dbo.queries WHERE id = ?"
    curs = conn.cursor()
    curs.execute(stmt, str(r))
    for q in curs:

```

```

        query = list(q)
        curs.close()
        return query

def runRandomQuery(conn, query):
    stmt = query[0]
    curs = conn.cursor()
    print(stmt)
    curs.execute(stmt)

qconn = getConnection('tpcc_queries')
mconn = getConnection('tpcc')
delayS = 0
durationS = 300

def main(qconn, mconn):
    query = getRandomQuery(qconn)
    runRandomQuery(mconn, query)
    time.sleep(delayS)

startTime = time.time()
while time.time() <= startTime + durationS and successCount <= 1800:
    try:
        main(qconn, mconn)
        successCount += 1
    except:
        print('----- Failed! -----')
        failedCount += 1

print('Success count: ' + str(successCount))
print('Failed count: ' + str(failedCount))

```

END OF DOCUMENT.