Journal of Database Management
Volume 31 • Issue 4 • October-December 2020

Investigating the Effects of Object-Relational Impedance Mismatch on the Efficiency of Object-Relational Mapping Frameworks

Derek Colley, Staffordshire University, UK
Clare Stanier, Staffordshire University, UK
Md Asaduzzaman, Staffordshire University, UK

ABSTRACT

The object-relational impedance mismatch (ORIM) problem characterises differences between the object-oriented and relational approaches to data access. Queries generated by object-relational mapping (ORM) frameworks are designed to overcome ORIM difficulties and can cause performance concerns in environments which use object-oriented paradigms. The aim of this paper is twofold, first presenting a survey of database practitioners on the effectiveness of ORM tools followed by an experimental investigation into the extent of operational concerns through the comparison of ORM-generated query performance and SQL query performance with a benchmark data set. The results show there are perceived difficulties in tuning ORM tools and distrust around their effectiveness. Through experimental testing, these views are validated by demonstrating that ORMs exhibit performance issues to the detriment of the query and the overall scalability of the ORM-led approach. Future work on establishing a system to support the query optimiser when parsing and preparing ORM-generated queries is outlined.

KEYWORDS

Database Performance, Object-Relational Mapping (ORM), Performance Tuning, Query Performance, Relational Databases, Structured Query Language (SQL)

INTRODUCTION

Object-relational impedance mismatch (ORIM) occurs when object-oriented application development – a hierarchical paradigm - meets the relational database layer, a set-based paradigm. ORIM is categorised into several layers of granularity, from concept to language (Chen et al., 2014; Ireland et al. 2009). In implementation terms, ORIM means overcoming the mismatch between invoking a method within an application to generating the Structured Query Language (SQL) that is required by the method. Using inline SQL can meet this need, but this method is intolerant of schema changes and can introduce security flaws, such as injection. Using a stored procedure layer as an alternative can help mimic the object-oriented model in the database, but comes at the cost of moving the application logic into the data layer, tightening the coupling between these two layers, potentially moving the logic out of source control and necessitating SQL skills to make changes in the future. The need to address relations as objects in the application-database interface instead bred a third solution, a class

DOI: 10.4018/JDM.2020100101

of tools known as object-relational mapping (ORM) frameworks, designed to bridge the gap between object-oriented method calls and the generation of Structured Query Language (SQL) queries. ORM frameworks differ in specifics, but typically store an internal data model and use rule bases and heuristics to generate SQL from this data model in response to application requests. The resultant SQL is presented to the relational database management system (RDBMS).

Relational databases are data stores that operate according to the long-established principles of relational algebra (Date, 1990; Astrahan et al., 1976; Held, Stonebraker & Wong, 1975; Codd, 1974; Stoll, 1963). In contrast to document-style databases, storing unstructured or semi-structured attribute-value pairs, relational database design is based on relations, or sets of tuples of related values, which are stored in tables, linked with keys and queried with SQL. It is claimed that 4 of the top 5 most popular database tools in recent use are based on the relational model (Solid IT, 2018).

This paper investigates whether ORM frameworks are well-suited to producing SQL queries, given that ORM tools are, themselves, object-oriented constructs; more specifically, whether the mismatch postulated by ORIM can be observed in the methods and outputs of ORM tools as measured by relative performance; in essence, are ORM tools producing efficient SQL?

We investigate this aim in three ways: firstly, through a literature review of ORIM and associated relevant topics; secondly, through the administration of a survey of practising database professionals aimed at gathering expert opinions on the perceived effectiveness of ORM tooling, using thematic analysis to construct appropriate narratives; and thirdly by investigating, through empirical experimentation and using industrial software, the operation of ORM-generated performance impacts on a relational database using a benchmark data set, extending our prior research (Colley, Stanier & Asaduzzaman, 2018) in this area. By combining all three approaches, we validate whether the opinions of our survey participants are borne out by the findings of the ORM testing; whether the results of the ORM testing concord with the findings of other researchers; and to establish whether ORIM is a solved problem through the use of ORMs, or whether ORIM at the application-database interface remains a current and relevant issue, to be addressed by future research.

The remainder of this paper is structured as follows. The Literature Review section defines ORIM in more detail, summarises prior research into the issue, and describes how the issue can be manifested in relational database systems. The Problem Investigation section describes the investigation; split into two sections, the Domain Expert Views sub-section explains the methodology and process of gathering domain-expert views and presents the results; and the Empirical Investigation sub-section describes the experimental investigation into the impacts of ORM-generated queries and the results of this work. The Conclusion section draws together these results in the context of existing research, and Future Work discusses ideas for further research in mitigating ORM-generated query performance issues and future strategies for addressing ORIM in the data layer.

LITERATURE REVIEW

The need for a mechanism to translate from object-oriented programming methodologies to the relational database layer is relatively new, gaining momentum since only the mid-1990s (compared to the development of database platforms from the late 1960s). Prior to the advent of object-oriented programming, functional programming, characterised by linear program flow and fixed application code, allowed for SQL to be written as part of the application. As object-oriented programming became more prevalent throughout the 1990s, researchers started to recognise that what would become known as object-relational impedance mismatch (ORIM) was a real concern when using a relational data layer, and started to create solutions – precursors to today's object-relational mapping (ORM) frameworks – to address the issue. Durand et al. (1994) were one of the first to do so, with the 'Object View Broker' – a primitive ORM that stored object-oriented views of relational data; in the same year, Kemp et al. (1994) experimented with storing object-oriented data in relational data stores, and created what they termed an 'object/relational mapping' methodology to do so, based on object-oriented data architecture (Kim, 1990; Banerjee, 1987). This decade proved to be the beginning of the development of ORM tools; Orenstein (1999) presents an ORM built for Java, predating Hibernate; Jungfer et al. (1999) note how coding 'Interface Definition Language' components to communicate with relational databases is 'tedious' and propose a new language and structure to effect this semi-automatically; Mlynkova and Pokorny (2004) are among the first to consider the use of XML as a data mapping layer between the application and the relational layers (a method used today in some ORMs).

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. Karwin (2017) labels some of these issues 'anti-patterns'; these are undesirable behaviours of the query or queries that exhibit in several different ways. In effect, these anti-patterns are manifestations of object-relational impedance mismatch in the 'paradigm' and 'language' classes (Ireland et al., 2009).

Karwin discusses SQL anti-patterns in general but specifically identifies issues with ORM-generated queries. Models (in the Model-View-Controller (MVC) arrangement) are very closely coupled with database schemas; 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 in the applications.

It was demonstrated by Chen et al. (2014) that these anti-patterns, later identified by Karwin (2017), 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 in that the data 'in-hand' in each execution loop is a small subset of the whole, from a database performance perspective this can produce an unwanted number of table or index lookups (or table/index scans, or index seeks) and can lead to an exponential overhead in query processing time and resource consumption. By the tenets of relational theory, it is accepted that set-based queries are preferred over object-based accesses due to better efficiency and lower query cost (Karwin, 2018; Cheung et al., 2016; Fritchey, 2017; Chen et al. (2014); Date, 1990).

Chen et al. (2014) also describe the eager fetching problem ('excessive data') common to ORMs where extra columnar data is brought through to the application from within the query then discarded when the results are compiled, and demonstrated a 71% increase in performance for a set of queries when mitigating this anti-pattern through experimentation on a standard data set. Cheung et al. (2016) repeated this finding and reported the details of how ORMs can hide this behaviour from the user, for example by using pre-fetching to group SQL calls, with mixed results (Ramachandra and Sudarshan, 2012). The consequences of pre-fetching data can include slower execution time, increased system resource use, and more data traffic - the manufacturers of ORM tools also report related adverse behavioural patterns with their tools; Microsoft Corporation (2009), the creator of Entity Framework, the dominant ORM tool in the .NET application stack, describe 8 different performance considerations 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.

Depending on perspective, the implementation of ORMs has been a mixed success. For application developers, ORMs can abstract away the maintenance of hardcoded SQL, simplifying development. Calling ORM-supplied methods rather than executing stored procedures or running inline queries is convenient and compatible with object-oriented programming languages and fits into the current *zeitgeist* of Agile, continuous-integration led application development. However, one important drawback of the model-based approach is the maintenance overhead involved in keeping the conceptual, or logical, data model and the physical data model in synchronicity, which can manifest in difficulties maintaining the code base (An, Hu & Song, 2010).

From the perspective of the database administrator, ORMs can present serious performance issues. Replacing static queries with dynamically-generated queries has inherent problems; the aforementioned N+1 and eager fetching problems (Microsoft Corporation, 2009; Astrahan et al., 1976); larger execution plans, reducing the effectiveness of the plan cache; excessive recompilations due to lack of parameterisation; the promotion of less well-performing structures like nested queries at the expense of set-theoretic constructions such as JOINs; the avoidance of advanced language constructions which could aid efficiency, such as window functions; and difficulty diagnosing access patterns (He and Darmont, 2005).

Fundamental construction issues at the heart of ORM development that stem from the formal classification of object-relational impedance mismatch continues to be a contemporary issue. Torres et al. (2017) present a survey of nine different types of ORM tool and relate the characteristics of each tool to the underlying design patterns in the literature. Their survey has some notable outcomes; first, that every tool they assessed was found to be implementing strategies to mitigate the ORIM problem (which they term IMP – Impedance Mismatch Problem) and that those strategies map to logical design patterns. This implies that there is no single ORM product which is significantly better or worse at dealing with ORIM, only that each product uses the same underlying methodologies to implement a partial solution, which in turn implies that a complete solution has yet to be found. Although they explicitly exclude the ability of the ORM to generate SQL queries from their assessment, their findings show that ORM tools are merely implementations of a common set of techniques to work around ORIM, meaning that queries that result from such tools will share the same characteristics preventing them from being completely efficient against database schemata.

There is also evidence in the literature that practitioners are still finding ORM tools can be unfit for purpose. Vial (2018) defines ORMs and highlights lessons learned about their effective implementation. Many of his findings echo earlier literature (ORMs have fetching problems which bring about the N+1 pattern, as per Astrahan et al., 1976); and the suggested mitigations include deferring logic back to the database in the form of computed columns and stored procedures, or overcoming the problem by moving databases in-memory. This is an acknowledgement that from a purist perspective, ORMs are not providing satisfaction in the field since the mismatch issues continue to be felt and strategies developed to work around the problems that ORMs present.

ORMs, however, continue to be a popular tool irrespective of performance problems. Ismailova & Kosikov (2018) go so far as to suggest that the relational element of the object-relational mismatch could be at fault; that ORMs lack an overarching conceptual basis that is not rooted in the relational model, and that a reassessment of the relational model (moving from Boolean algebra in set theory to Stone algebra) might be in order for ORMs to become more generalised. Although this appears to be a singular view, some responsibility for mitigating negative performance effects of ORMs and ensuring queries can be executed in a timely and efficient manner must also fall upon the relational database management system. During query processing, queries are disassembled, or parsed (Pachev, 2007), bound to objects, arranged into an execution plan and executed against a base schema. Although the optimiser is able to mitigate some effects of inefficient SQL query constructions through the simplification of queries to a parse tree and a collection of heuristics, it is evident through these repeated findings in the literature that the cost-based optimiser was not designed to deal with the query structures and sub-optimal performance behaviours associated with ORM-generated queries.

An alternative approach is to modify the relational schemas to fit the demands of the ORM-generated query, rather than attempt to modify the queries to fit the underlying relational schemas. Bolloju (1997) was an early identifier of this approach, and investigated whether schema denormalisation was an effective method for better object-relational interaction. Denormalisation is the process of rearranging a logical database schema to reduce the number of tables and so facilitate queries with fewer JOINs – for ORMs, which tend to produce nested queries rather than use JOIN syntax, this is beneficial for performance, but can lead to undesired outcomes; Bolloju (1997) identifies excessive fragmentation, but arguably this proceeds from the physical storage of the data, rather than the logical; but also identifies issues with integrity (maintaining data integrity in denormalised tables which allow duplicates is problematic) and flexibility (normalised database schemas are built to scale well). Data clustering is proposed as an alternative, where data is rearranged to better resemble the object-relational hierarchical model of data. There is evidence of research into the logical mapping of 'metaschemas' – methodologies for schema creation – in both the object and the relational domains (x, x) which provide the conceptual foundation for remapping schemas (Halpin, 2002).

He (2005) identifies changing access patterns in databases as a driver for performance issues. This is particularly prevalent in queries generated by ORMs, which by their nature can be unpredictable, generated automatically and prone to change between iterations or generations of subsequent queries. Several RDBMS platforms support plan caching; this is defined as the storage of the execution plan for a query for subsequent re-use (removing some of the cost of plan generation) (Microsoft, 2019) and is achieved by the substitution of string literals for parameters. ORM-generated queries can fill an execution plan cache through the over-generation of similar query plans that cannot be grouped in this fashion and as such fill the cache, necessitating the generation of new plans on every execution not just for the ORM-generated queries in hand but for other queries being run on the same database instance.

The future for ORMs as a data layer between object and relational paradigms is unclear. Although the tools are constantly improving, the anti-patterns associated with ORM tooling persist. In the remainder of this paper we seek the expert opinions of database practitioners on the effectiveness of ORM tools, and experiment upon a benchmark data set to attempt to replicate some of the issues reported in the literature on industry-standard tools.

PROBLEM INVESTIGATION

The investigation of the problem was split into two halves; first, through a survey of database practitioners with a focus on the uses of ORM frameworks, performance tuning in database environments, and the future of relational database query performance tuning. Secondly, an experiment to determine the performance outcomes from a series of scenario-based queries was designed to compare and contrast the relative performance of queries written by a specialist and queries produced by an ORM tool. The results of both investigations are presented in this section.

Domain Expert Views

Given prior research into object-relational impedance mismatch, 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.

In this section, evidence is sought as to whether ORM-produced queries, and ORMs in general, cause performance issues in real-life database environments. A survey consisting of 18 questions for an audience of database practitioners was designed, piloted and delivered 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.

Design

The survey was designed to capture results using a mixed-methods approach. The questions were structured primarily using Likert-scaled questioning, 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 a number of database-specific statements on a 5-point Likert scale with an additional neutral option added (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. Responses were analysed as indicative samples of opinion using qualitative analysis, with free-text commentary from the respondents treated as significant and central contributions. As an alternative, the methodology of thematic analysis (Clarke & Braun, 2013; Aronson, 1995) 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. In mitigation, 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-checking was designed to measure validity, and it was found to be effective during analysis of the resulting data with few contradictions in the results.

Additionally, 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. 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.

Results and Analysis

There are several commonly-accepted stages of thematic analysis as defined by Clarke and Braun (2013), 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, 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 in order 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 1 shows the outcome of this phase.

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. Figure 1 shows the thematic map with themes as ellipses, sub-themes as rounded rectangles, and the links and insights associated with them.

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.

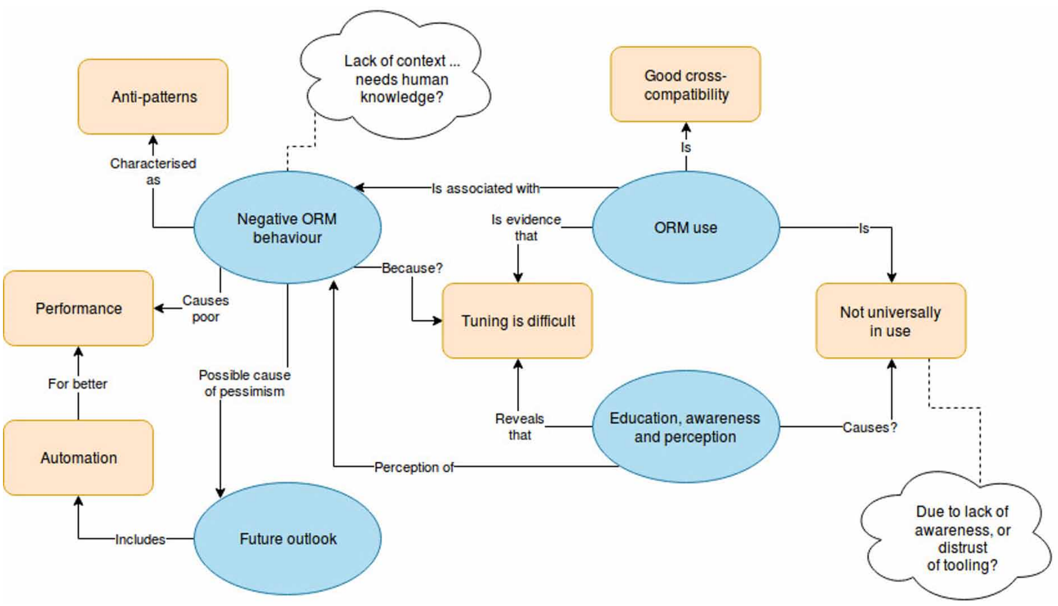
Theme - ORM Use

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).

Table 1. Final codification of the survey data

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

Figure 1. Thematic map of survey results



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. (2009). The difficulties of tuning ORMs is 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 (Ambler, 2018; Ambler, 2008), 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 (Niu, Martin & Powley, 2009). 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.

In conclusion, the survey indicated that ORMs are distrusted among database practitioners; that there is a perception, backed by anecdotal evidence, that ORM tools create performance tuning problems; that there is an appetite for more automation in performance tuning and database management; that practitioners felt there is a lack of awareness among developers around effective ORM use; and that ORM uptake is significant enough in industry for ORM-generated query tuning to be an important and timely research issue.

Next, an investigation was carried out into whether performance issues reported by the survey can be replicated through experimentation with an industry-standard ORM tool.

Empirical Investigation

The purpose of the experimental validation is to triangulate upon the findings of the survey, particularly around the finding that practitioners experienced performance issues when dealing with ORM-generated queries. We investigate 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 (Pacific Marine Environmental Laboratory, 2018) 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 (Bay et al., 2000), ensuring repeatability. 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 (Karwin, 2017; Ireland et al., 2009) 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 (Fritchey, 2017). 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 (2017) and summarised in Table 2.

Table 2. 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 described in Tables 3 to 7 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.

The non-ORM generated queries were written manually by a subject matter expert 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 8 shows how the manual SQL (non-ORM) queries compare with the ORM-generated queries for the 7 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.

Table 3. 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	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

Python/Django	FactTAO.objects.all().select_related('datekey').values('datekey__mthnum', 'datekey__yrnum'). annotate(Avg('airtemp')).order_by('datekey__mthnum', 'datekey__yrnum')

ORM SQL	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

Table 4 Query Objective O2

Descriptor	Values
Summary	Return the earliest and latest dates for which buoy sensor readings exist within the data set.

Manual SQL	SELECT MIN(f.dateKey) [earliestDate], MAX(f.dateKey) [latestDate]
	FROM dbo.factTAO f

Python/Django	FactTAO.objects.aggregate(Min('datekey'), Max('datekey'))

ORM SQL	SELECT MIN([factTAO].[dateKey]) AS [datekey__min],
	MAX([factTAO].[dateKey]) AS [datekey__max]
	FROM [factTAO]

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.

Table 5. Query Objective O3

Descriptor	Values
Summary	Return the latitude and longitude positions of all buoys in January 1984, with no ordering.

Manual SQL	SELECT f.obsID, 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.yrNum = 1984 AND d.mthNum = 1

Python/Django	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)
ORM SQL	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)

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.

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. (2009)), 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.

Table 6. 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	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

Python/Django	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:
	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'

ORM SQL	(@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

Table 7. 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	;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)
	SELECT TOP 1 CAST(d.d AS NUMERIC(16,2)) [MaxDistance]
	FROM distances d
	ORDER BY [d] DESC

Python/Django	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)

ORM SQL	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)',997,997,46.064999999999998,46.064999999999998,57.380000000000003,57.380000000000003 select @p1 SELECT [locationDataTempTable].[uqid], [locationDataTempTable].[from], [locationDataTempTable].[to], [locationDataTempTable].[fromLat], [locationDataTempTable].[toLat], [locationDataTempTable].[fromLong], [locationDataTempTable].[toLong] FROM [locationDataTempTable]

Table 8. 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

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 ('Spherical Trigonometry', n.d., para. 6). 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

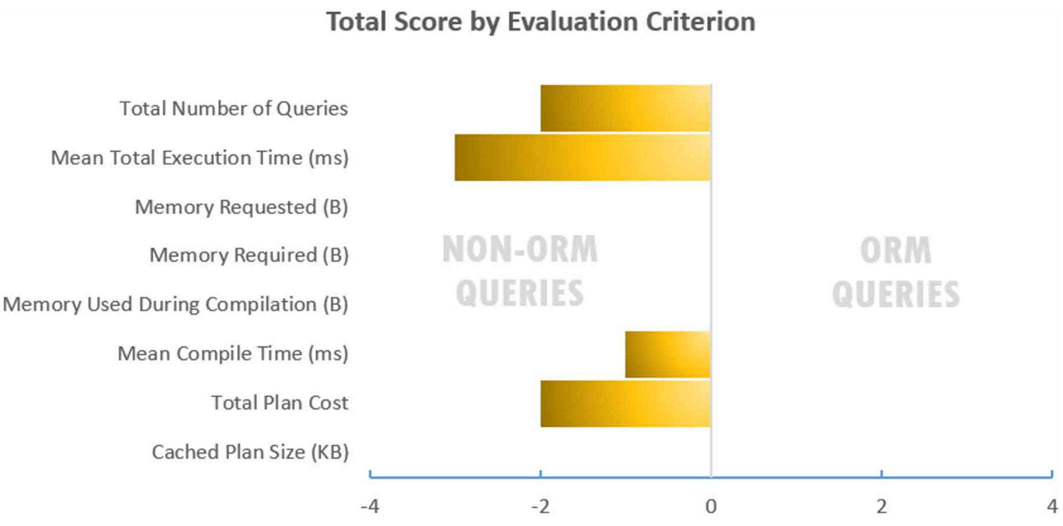
The results are assessed against the evaluation criteria as follows. For each criterion, the two results – for the ORM-generated query, and for the non-ORM generated query – are compared using the condition for the criterion specified in the 'Comparative Rule' column (of Table 2). If the non-ORM result 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, as per Table 9, 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, we assign the scores 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 Figure 2, 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.

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

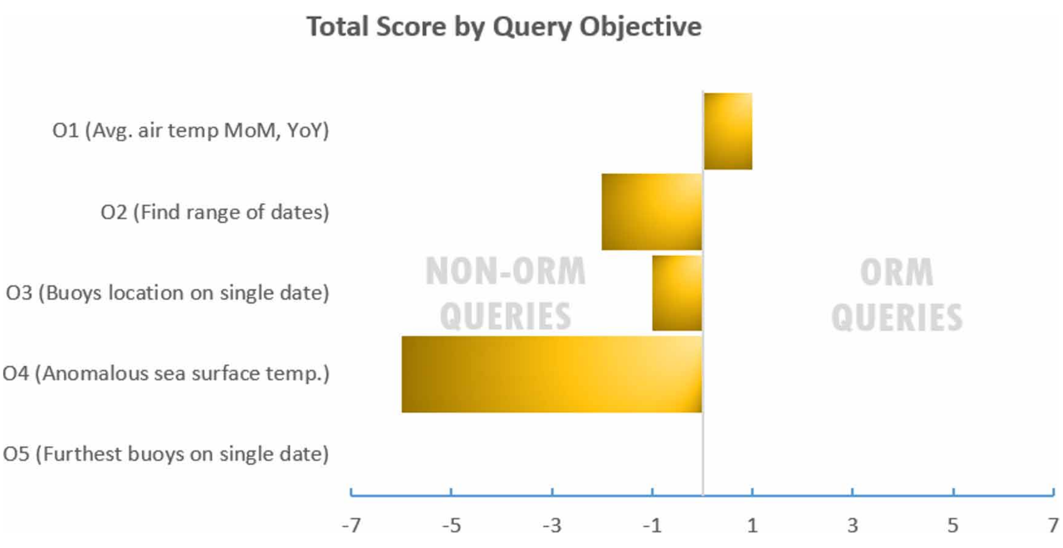
Figure 2. Total score by evaluation criterion



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.

Figure 3. Total score by query objective



The scores for the query objectives across all evaluation criteria are illustrated in Figure 3. 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.

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. Figure 4 illustrates

Figure 4. Correlation between increasing complexity and execution time of ORM methods

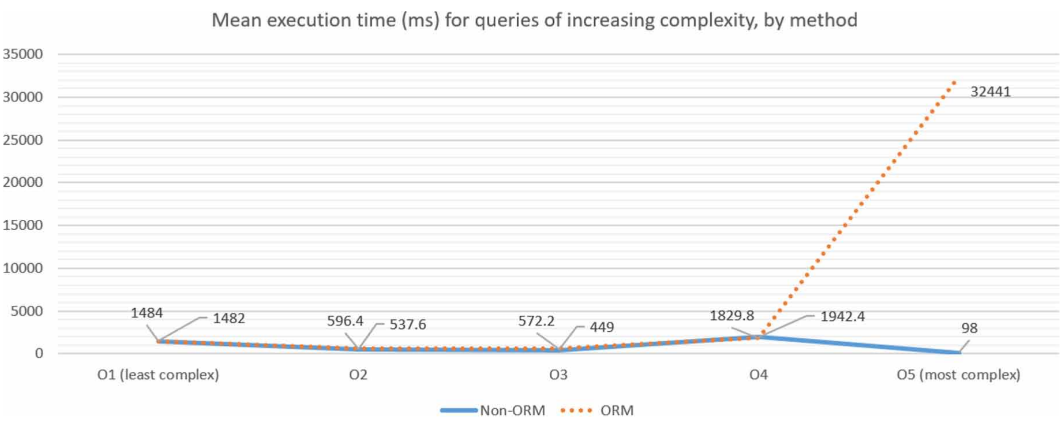


Table 9. p-values from t-testing of mean execution time observations

	Non-ORM	ORM
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**	

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).

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 illustrated in Figure 2, this analysis is borne out by the *p*-values obtained for all the observations. Table 9 shows that in these *t*-tests, $p > 0.05$ (highlighted). This supports 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.

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 (Karwin, 2017; Chen et al., 2014) and observed in related studies (Colley, Stanier & Asaduzzaman, 2018; Colley & Stanier, 2017). The *p*-value testing of the results of one important evaluation criterion helped support the evidence ($p > 0.05$) that this correlation exists.

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.

RESEARCH CONTRIBUTIONS

Object-Relational Mapping tools are ubiquitous in modern application development environments, with a range of different packages available for most contemporary development languages and frameworks. The necessity of these packages is due to the need for object-oriented languages to communicate with the relational model. ORMs are able to provide practical SQL queries in response to method calls, and to translate received results to objects for consumption by the application layer, but in our literature review, we questioned the suitability of ORMs to provide optimised SQL queries, performant to the same degree as well-written SQL queries executed directly against the database platform. We identified several important works in the literature that acknowledged this problem and characterised some attempts at mitigation. The importance of ascertaining the efficiency of ORM-generated SQL and finding strategies to mitigate the generation of poorly-performing queries is of paramount importance in the near future; as the velocity, variety and volume of data continues to grow, queries must deal with an increasing level of information density and to continue to operate in any practical manner, the time is right for a re-examination of the fundamental paradigms behind ORM software tools. Our research aims to contribute to the body of literature that has already investigated this area and show whether these problems are still current in modern ORM environments, both through survey of the practitioners involved and demonstration using current tools, and to work to identify ways in which the object-relational impedance mismatch problem can be solved.

In our research, through questioning database practitioners about their experiences with ORM tooling, we found the majority of practitioners take a dim view of their efficacy. Our survey results add to the growing body of research that noted the disparity between object-oriented and relational coding (Jungfer et al., 1999) through to the formal classification of the problem (Ireland et al., 2009), and more recently cataloguing the design flaws inherent in such systems (Chen et al., 2014; Karwin, 2017, Torres et al., 2017). There is comparatively little qualitative research available which questions industrial database practitioners specifically about this issue, and so our findings add some credence to views elsewhere in the literature (ibid.) that ORMs are not entirely well-suited as solutions to the object-relational impedance mismatch problem in the field.

We acknowledge there may exist some latent flaws in our survey research; notably, the relatively low respondent count, and the potential reliability of the respondents' answers brought about by their self-selection as participants. These are not original problems with survey research in general, but difficult to overcome considering the relatively specialist audience. We sought to mitigate these issues by investigating the problem further using experimental work, using one primary outcome from the survey –it is perceived that ORM queries tend to be less performant than manually-written queries. In doing so, we tested the currency and validity of this claim, with some success. Our theoretical contributions are therefore the confirmation that the object-relational impedance mismatch problem is still current; that implementation of ORMs do not yet fully mitigate its theoretical effects; and that the perception of ORMs in the industry is poor, meaning more work in better integrating the object and relational worlds to produce more effective solutions is likely to be beneficial in future industrial software development.

Examination of the underlying conceptual and practical issues of ORMs, whether by qualitative or quantitative means, is not new; numerous examples by Karwin (2017) and Torres et al. (2017) could be seen as more thorough; however our investigations and subsequent results from both strands benefit from being industry-aligned, less abstract than purely academic studies and rooted in concrete, reproducible outcomes. Consequently, our practical contributions to the field are the identification of specific use-case patterns where SQL queries are better-performing through direct calls from the application layer than through an ORM; the re-affirmation that this issue continues to be current; and the conclusion that ORMs have negative performance implications as query complexity increases, both in terms of material, measurable performance impacts to the calling application and system resources through the display of inefficient design patterns, which may encourage mitigation strategies such as the design of database schemas for simplicity.

CONCLUSION AND FUTURE WORK

We analysed the survey results thematically and drew a number of narratives from the findings. These narratives were characterised as *ORM use*; *education*, *awareness and perception*; *negative ORM behaviour*; and *future outlook*. The survey results suggested that ORM tools are not ubiquitous but are present in a sizeable minority of respondents' organisations. It was felt by respondents that ORMs were fundamentally incompatible in several ways with RDBMS systems; that they were difficult to tune, and this was supported by examples from the respondents that were echoed in the literature (ibid.). The findings suggested that at least some of the performance difficulties associated with ORM tooling can be attributed to lack of awareness in the developer community in how to use these tools efficiently, although this conclusion is arguably countered by the differences of opinion between the database administration community and the developer community (Ambler, 2018; Neward, 2006). The survey findings were supportive in general of automation and positive about the future of relational database performance tuning, albeit sceptical of the role that ORM tooling may play in that future.

The exploration of the objectives was continued in the experimental investigation. Five query objectives were defined and based on a real data set were constructed and presented in increasing order of complexity, and the performance of ORM and non-ORM generated versions compared. The outcome of our experimentation showed that when the results were grouped by performance metric, manually-written (non-ORM) queries outperformed ORM-generated queries. This meant that after assessing each query by each evaluation criterion and producing sum totals grouped by criterion *in no cases did ORM queries generally outperform manually-written queries*; this only occurred when considering individual results, and for low levels of complexity. The findings were then pivoted to analyse the evaluation criteria by the queries in increasing order of complexity. Using this view of the data it was shown that in 3 of 5 cases, manually-written queries outperformed ORM-generated queries, with 1 *vice versa* case and 1 inconclusive observation. Deconstructing the outcomes by query complexity, it was found there was a positive correlation between query complexity and query execution time and that the ORM-generated queries consistently took longer to run ($p = 0.18$) than non-ORM queries as complexity increased.

Additionally, undesirable behaviour was observed by the ORM tooling and mirrored some of the behaviour listed by the respondents of the survey and detailed in the literature. In particular, redundant code, lack of support for the full language, multiple queries, implicit conversion and row-by-row processing (the N+1 problem) were observed.

The survey findings and experimental evidence support the conclusion that ORM tooling has negative performance implications as query complexity increases, both in terms of material, measurable performance impacts to the calling application and system resources, and through the display of inefficient design patterns. ORMs are widespread but not universally used with the survey findings suggesting that barriers to adoption are related to perceived poor performance, and so there is a gap for future research into approaches to mitigate or remove the negative impacts to database query performance caused by ORM tools.

Our findings have illustrated the need to consider alternative approaches to database performance tuning that are better able to assist in compiling and executing queries generated from non-traditional sources such as ORM frameworks. Such a solution could take the form of a flexible database performance management framework capable of handling sub-optimal queries.

One potential solution is a model incorporating some degree of schema selectivity. One of the issues noted in the findings of this paper was the existence of redundant code and the existence of implicit conversion problems. Other undesirable behaviours, such as fetching more columns than required, suggest that a potential direction is the creation of multiple schemas within a database, each of which is essentially a 'whole-database' index. Building alternative schemas on the same tree-based principles as indexes but widening the scope to multiple objects could form the basis of a new schema selection algorithm. This is not without precedent, as Chen (1999) investigated the choice of alternate schemata on a query-by-query basis with initially positive findings. One direction we are exploring is the employment of a query comparison and schema selection algorithm using an alternative model for query representation. Another direction is the reconsideration of SQL queries as constructed objects composed of first-order logic, representable in more computable and comparable ways within database engines. The viability and detail of these proposed solutions are current research topics for the authors.

REFERENCES

Ambler, S. W. (2008). When it gets cultural: Data management and Agile development. *IT Professional*, *10*(6), 11–14. doi:10.1109/MITP.2008.135

Ambler, S. W. (2018). *The Cultural Impedance Mismatch Between Data Professionals and Application Developers*. Retrieved from http://www.agiledata.org/essays/culturalImpedanceMismatch.html

An, Y., Hu, X., & Song, I. (2010). Maintaining mappings between conceptual models and relational schemas. *Journal of Database Management*, *21*(3), 36–68. doi:10.4018/jdm.2010070102

Aronson, J. (1995). A pragmatic view of thematic analysis. *Qualitative Report*, *2*(1), 2–3. https://nsuworks.nova.edu/tqr/vol2/iss1/3/

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., Traiger, I. L., Wade, B. W., & Watson, V. (1976). System R: Relational approach to database management. *ACM Transactions on Database Systems*, *1*(2), 97–137. doi:10.1145/320455.320457

Banerjee, J., Chou, H. T., Garza, J. F., Kim, W., Woelk, D., Ballou, N., & Kim, H. J. (1987). Data model issues for object-oriented applications. *ACM Transactions on Information Systems*, *5*(1), 3–26. doi:10.1145/22890.22945

Baroni, M., Dinu, G., & Kruszewski, G. (2014). Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors. *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, *1*, 238-247. Retrieved from http://clic.cimec.unitn.it/marco/publications/acl2014/baroni-etal-countpredict-acl2014.pdf

Bay, D. S., Kibler, D. F., Pazzani, M. J., & Smyth, P. (2000). The UCI KDD Archive of Large Data Sets for Data Mining Research and Experimentation. *SIGKDD Explorations*, *2*(2), 81–85. doi:10.1145/380995.381030

Bolloju, N., & Toraskar, K. (1997). Data Clustering for Effective Mapping of Object Models to Relational Models. *Journal of Database Management*, *8*(4), 16–24. doi:10.4018/jdm.1997100102

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. Retrieved from: https://opencommons.uconn.edu/dissertations/AAI9942566/

Chen, C. M., & Roussopoulos, N. (1994). The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. *International Conference on Extending Database Technology*, *1*, 323-336. doi:10.1007/3-540-57818-8_61

Chen, T., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2014). Detecting performance anti-patterns for applications developed using object-relational mapping. *Proceedings of the 36th International Conference on Software Engineering*, 1001-1012. doi:10.1145/2568225.2568259

Cheung, A., Madden, S., & Solar-Lezama, A. (2016). Sloth: Being lazy is a virtue (when issuing database queries). *ACM Transactions on Database Systems*, *41*(2), 8. doi:10.1145/2894749

Clarke, V., & Braun, V. (2013). Teaching thematic analysis: Overcoming challenges and developing strategies for effective learning. *The Psychologist*, *26*(2), 120-123. Retrieved from http://eprints.uwe.ac.uk/21155

Codd, E. F. (1974). Recent Investigations into Relational Data Base Systems. IBM Research Report RJ 1385. In *Proceedings of the 1974 Congress*. New York, NY: North-Holland.

Colley, D., & Stanier, C. (2017). Identifying New Directions in Database Performance Tuning. *Procedia Computer Science*, *121*, 260–265. doi:10.1016/j.procs.2017.11.036

Colley, D., Stanier, S., & Asaduzzaman, M. (2018). The Impact of Object-Relational Mapping Frameworks on Relational Query Performance. *Proceedings of the International Conference on Computer, Electrical and Electronics Engineering 2018 (ICCECE '18)*, 47-52. Retrieved from: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8659222

Date, C. J. (1990). *Relational database writings, 1985-1989* (Vol. 1). Addison-Wesley.

Durand, J., Ganti, M., & Salinas, R. (1994) Object View Broker: A mediation service and architecture to provide object-oriented views of heterogeneous databases. *Applications of Databases: Lecture Notes in Computer Science*, 819. Retrieved from https://link.springer.com/chapter/10.1007/3-540-58183-9_63

Fritchey, G. (2018). *SQL Server 2017 Query Performance Tuning*. Apress. doi:10.1007/978-1-4842-3888-2

Halpin, T. (2002). Metaschemas for ER, ORM and UML data models: A comparison. *Journal of Database Management*, *13*(2), 20–30. doi:10.4018/jdm.2002040102

He, Z., & Darmont, J. (2005). Evaluating the dynamic behavior of database applications. *Journal of Database Management*, *16*(2), 21–45. doi:10.4018/jdm.2005040102

Held, G. D., Stonebraker, M. R., & Wong, E. (1975). INGRES: a relational data base system. *Proceedings of the May 19-22, 1975 National Computer Conference and Exposition*, 409-416.

Ireland, C., Bowers, D., Newton, M., & Waugh, K. (2009). *A Classification of Object-Relational Impedance Mismatch*. First International Conference on Advances in Databases, Knowledge, and Data Applications. doi:10.1109/DBKDA.2009.11

Ismailova, L. Y., & Kosikov, S. V. (2018). Metamodel of Transformations of Concepts to Support the Object-Relational Mapping. *Procedia Computer Science*, *145*, 260–265. doi:10.1016/j.procs.2018.11.055

Jungfer, K., Leser, U., & Rodriguez-Tomé, P. (1999). Constructing IDL views on relational databases. *International Conference on Advanced Information Systems Engineering*, 255-268. Retrieved from https://link.springer.com/content/pdf/10.1007/3-540-48738-7_19.pdf

Karwin, B. (2017). *SQL Antipatterns*. Pragmatic Bookshelf.

Kim, W. (1990). Object-oriented databases: Definition and research directions. *IEEE Transactions on Knowledge and Data Engineering*, *3*(3), 327–341. doi:10.1109/69.60796

Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., & Byers, A. H. (2011). *Big data: The next frontier for innovation, competition, and productivity.* McKinsey Global Institute. Retrieved from https://www.mckinsey.com/~/media/McKinsey/Business%20Functions/McKinsey%20Digital/Our%20Insights/Big%20data%20The%20next%20frontier%20for%20innovation/MGI_big_data_exec_summary.ashx

Microsoft Corporation. (2009). *Getting Started with Entity Framework 6 Code First using MVC 5*. Retrieved from 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

Microsoft Corporation. (2019). *sys.dm_exec_query_plan_stats (Transact-SQL)*. Retrieved from https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-exec-query-plan-stats-transact-sql?view=sql-server-ver15

Mlynkova, I., & Pokorný, J. (2004). *From XML Schema to Object-Relational Database-An XML Schema-Driven Mapping Algorithm*. ICWI. Retrieved from http://www.cs.cas.cz/semweb/download/04-10-Mlynkova.pdf

Neward, T. (2006). *The Vietnam of Computer Science*. Retrieved from https://pdfs.semanticscholar.org/331e/490c55ee72d6011bbceb323c03f0572a5235.pdf

Niu, B., Martin, P., & Powley, W. (2009). Towards autonomic workload management in DBMSs. *Journal of Database Management*, *20*(3), 1–17. doi:10.4018/jdm.2009070101

Orenstein, J. A. (1999). Supporting retrievals and updates in an object/relational mapping system. *IEEE Data Eng. Bull., 22*(1), 50-54. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.9399&rep=rep1&type=pdf#page=52

Pachev, S. (2007). *Understanding MySQL Internals*. O'Reilly.

Pacific Marine Environmental Laboratory (PMEL), National Oceanic and Atmospheric Administration (NOAA). (2018). *El Nino Data Set*. Retrieved from https://archive.ics.uci.edu/ml/datasets/El+Nino

Ramachandra, K., & Sudarshan, S. (2012). Holistic optimization by prefetching query results. *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 133-144. doi:10.1145/2213836.2213852

Solid, I. T. (2018). *DB-Engines Ranking*. Retrieved from https://dbengines.com/en/ranking

Spherical Trigonometry. (n.d.). *Wikipedia*. Retrieved November 23, 2018, from https://en.wikipedia.org/wiki/Spherical_trigonometry

Stoll, R. R. (1963). *Set Theory and Logic*. W H Freeman and Co.

Torres, A., Galante, R., Pimenta, M. S., & 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*, *82*, 1–18. doi:10.1016/j.infsof.2016.09.009

Vial, G. (2018). Lessons in persisting object data using object-relational mapping. *IEEE Software*, *36*(6), 43–52. doi:10.1109/MS.2018.227105428

Derek Colley is a researcher and lecturer in data management at Staffordshire University, and maintains a professional practice as an independent database consultant with more than a decade of extensive industry experience. He holds M.Sc and B.Sc degrees in Computer Science, a Postgraduate Diploma in Information Security and Digital Forensics and will complete his Ph.D in relational database optimisation in 2020. His research interests include data management, information representation theory, database administration and set theory with emphasis on the relational model. His current research project is the improvement of database query processing techniques through alternative representation forms.

Clare Stanier (PhD) was awarded a PhD in Information Systems from Staffordshire University in 2009. Dr Stanier researches and publishes in the field of Data Analytics and has a particular interest in the applications of machine learning to Business Intelligence and Big Data and Big Data Analytics. She is based at Staffordshire University where she lectures in Data Analytics and Data Management and supervises a number of PhD students working in these fields. Her major current research project is an investigation into the use of Big Data by Small and Medium Enterprises (SMEs).

Md Asaduzzaman received the B.Sc. and M.Sc. degrees in applied statistics from the University of Dhaka, Dhaka, Bangladesh, the M.Sc. degree in bioinformatics from Chalmers University of Technology, Gothenburg, Sweden, and the Ph.D. degree in operational research from the University of Westminster, London, U.K. He is a Senior Lecturer of statistics and operational research with Staffordshire University where he has been a faculty member since 2014, formerly a Lecturer and Assistant Professor of applied statistics with the Institute of Statistical Research and Training, University of Dhaka. His primary research interests include queueing, other stochastic models and mathematical programming for performance measure, capacity and resource planning, and management in healthcare, and telecommunication and other communication networks. He is also interested in statistical computing, large-scale data mining, and analysis in the earth, environmental sciences, and healthcare.