The Impact of Object-Relational Mapping Frameworks on Relational Query Performance

Derek Colley
Sch. Comp. & Digital Technologies
Staffordshire University
Stoke-on-Trent, United Kingdom
derek.colley@research.staffs.ac.uk

Clare Stanier
Sch. Comp. & Digital Technologies
Staffordshire University
Stoke-on-Trent, United Kingdom
c.stanier@staffs.ac.uk

Md Asaduzzaman
Sch. Creative Arts & Engineering
Staffordshire University
Stoke-on-Trent, United Kingdom
m.asaduzzaman@staffs.ac.uk

***Abstract*—This paper considers the impact of object-relational mapping (ORM) tools on relational database query performance. ORM tools are widely used to address the object-relational impedance mismatch problem but can have negative performance consequences. We first define the background of the problem, describing the growth of ORM tools against a backdrop of the changing application development landscape, then demonstrate examples of undesirable query performance patterns resulting from the application of these tools using a leading application stack. We review selected literature for prior research into this problem and summarise the findings. Finally, we conclude by suggesting future research directions to help mitigate the issues found with ORMs and signpost some potential solutions.**

Keywords—database, object-relational, query performance, object-relational impedance mismatch, ORM, relational, NoSQL.

I. Background

Data is all around us, and intrinsic to many of our interactions with the world. According to IBM [1], more than 1.5 x 10^9 GB of new data is generated every day, and at the time of writing 4 out of 5 database systems – repositories for this data – are of a relational kind [2]. Consequently, the care and maintenance of relational database management systems (RDBMSs) remains a current issue, especially in this era of the proliferation of big data [3], generated by phenomena such as the growth of social media and the emergence of the so-called Internet of Things [4].

Database engines have advanced significantly since the initial development of early systems like INGRES [5]. Tailored for functional application development patterns, most of the effort in developing early applications occurred during the design stage. Queries were then developed based on the known schema and could be contained within code modules such as stored procedures.

Application development largely followed the waterfall lifecycle [6] and so application releases were relatively infrequent. Consequently, query performance tuning took place either as part of the ordinary software development lifecycle (SDLC) or as part of day-to-day database maintenance activities. Once a query was tuned to the schema, further changes were limited.

Today, the software development lifecycle has developed beyond this functional, static model driven by the waterfall process to include object-oriented, dynamic models driven by SDLC paradigms such as Agile methodologies [6, 7]. The shift from infrequent to frequent, and even intra-day, software updates following the Continuous Integration/Continuous Development (CI/CD) [8] Agile approach, combined with the movement to object-oriented programming languages (OOPL), has contributed to a major issue with the integration of the relational data model with modern application development practices, usually referred to as the *object-relational impedance mismatch*. There is a substantial presence of material on this topic in the literature [9, 10, 11, 12, 13, 14, 15].

Previous research coverage of object-relational impedance mismatch has focused on the definition and categorisation of the problem [13], or on the merger of both concepts [12, 13] by augmentation of the relational model with object-oriented structures. The need for new directions in database performance tuning is discussed in [11] in response to this changing landscape.

This paper aims to build on this prior research by discussing the impact on database performance of ORM tools developed to bridge the object-relational gap. We also provide supporting examples from both the practitioner and academic literature of this behaviour and demonstrate selected examples. Given that these tools are now permanent features in the software architecture landscape, we suggest some techniques to mitigate the performance impacts and maximise database performance. Finally, we discuss potential future research directions in this area.

II. Object-Relational Mapping Frameworks

Object-Relational Mapping Frameworks (ORMs) are software solutions to the object-relational impedance mismatch problem. Although their implementation varies, the key idea is to provide a mapping from the object layer, such as an object as an instance of a class, to the relational, such as a collection of rows in a table. The differences between the object-oriented and the relational approaches are legion; objects have properties that preclude straightforward compatibility with databases. To illustrate these differences: objects are temporary, instantiated at runtime whereas data in the relational model is persistent; objects have no fixed schema, in contrast to a database; and a special, non-object-oriented language (SQL) is used to interact with the relational database, unlike OOPLs which have native syntax for standard method calls.

ORM tools cross the divide between object and relation by supplying an interface to the OOPL, which can be called as a method, and interacting with the database by generating and executing SQL. Thus, the application itself requires no capability for direct SQL manipulation, only the ability to call the methods supplied by the ORM tool. The ORM tool accomplishes this through the construction of an internal data model, the type of which varies according to the implementation, and database calls from the application are translated through this internal data model before execution.

III. Surveying the Object-Relational Divide

Ireland et al. [13], in their seminal paper, seek to describe the object-relational impedance mismatch and identify four levels of problem – the paradigm, the language, the schema and the instance, before discussing the issues that the ORM, as a solution, must address at each of these levels. These differences are defined as a range of incompatibilities between an object and a relation. This includes observations that an object is variable, encompassing both behaviour and state, with an identity unconnected to the attributes, in contrast to a relation which is a stateful object inclusive of an identifier (normally, but not always, the primary key). However, Ireland et al. [13] constrain their observations to construction of a theoretical framework to characterise object-relational impedance, in essence categorising the causes but not the effects.

Date and Darwen [12] take a different approach by suggesting that the theoretical relational model already has the requisite built-in logic to be compatible with object orientation, in that the theory supports domains (somewhat synonymous with types), and strong typing with user-defined types, which can themselves be rows of data. The authors argue that the principle of inheritance is supported in this way; additionally, that these so-called relational variables inherently carry their domain information. However, they contend that the industry has taken the wrong approach by pursuing the implementation of relations without the inclusion of metadata such as domains/types ('the first Great Blunder'), equating relations to object classes rather than domains to object classes and thus as it stands, the object relational impedance mismatch remains an issue.

Object-relational impedance mismatch remains a timely and relevant problem. The current zeitgeist towards the use of unstructured data sources could be regarded as symptomatic of the lack of cohesion between the object and relational worlds, emerging principally as a workaround to development issues caused by using fixed schemas and the SQL dialect. Unstructured data platforms have also found support amongst developers who find direct relational database access via drivers (such as ODBC and JDBC) in object-oriented applications cumbersome. An example of this is described in [16] where binding – the mapping between the object and the data object – cannot take place without an ORM, meaning the object model and relational models are essentially disconnected, causing diagnostic and inspection issues during code debugging since cause cannot be clearly established.

Un- or semi-structured ('NoSQL') databases are not solely used as workarounds for the avoidance of working with fixed schemas. Proponents of these data platforms argue that schemaless databases can store and serve data that does not fit the relational structure, such as streaming video and free text [17]. Without question, such data stores have a place and have found a closer fit to OOPL in many cases than the relational model, especially when dealing with data that does not require consistency. However, NoSQL databases cannot be the only answer to solving the object-relational impedance mismatch, as many of the benefits of the relational model are lost; relational integrity, the protections afforded by the ACID principles, the scalability afforded by normalisation, the standardised interface, and advanced indexing capabilities to name a few. As Vicknair et al. [18] unequivocally state, 'NoSQL rejects ACID'.

Consequently, we can establish that ORMs have a vital role to play in allowing developers to work with relational data sources in a way that is consistent with the class-object-property structure of OOPLs. However, ORMs have a set of endemic performance issues that arise as an artefact of their design. We can borrow the terminology of Karwin [14] to label these issues 'anti-patterns'.

From the practitioner's perspective, Karwin discusses SQL anti-patterns in general but specifically identifies issues with ORMs. According to Karwin, ORM tools are based on the Active Record design pattern which maps an object to a specific row in a table. One issue with this pattern is that models (in an MVC arrangement) are closely coupled with database schemas, so that if a schema changes the model becomes invalid. Another is that a class with the create, update and delete functionalities exposes these functionalities to any subclass inheriting from it, allowing direct database access, reducing cohesion.

Chen et al. [15] point out that ORM anti-patterns include row-by-row implementations as loops, especially when such structures are memory-efficient. In relational theory, set-based queries are normally preferred due to better efficiency and lower resource cost. Chen describes two patterns, the row-by-row problem and the eager fetching problem, termed 'excessive data', where additional columnar data is brought from the whole query and filtered in the application, and demonstrate a 71% increase in performance when this anti-pattern is mitigated. Cheung et al. [10] also discuss eager fetching, detailing methods of 'hiding' this from the user – including pre-fetching data. Pre-fetching data has other consequences than slower execution time, including increased system resource use. They present an approach named 'Sloth', which essentially groups queries in a query store, saving them up for execution in a single batch when the application flow forces a result.

The ORM manufacturers also recognise issues with ORM tools. Microsoft describe no less than 8 different performance considerations in Entity Framework [19] that negatively impact query performance (7 of which occur before or after the actual query execution). They also discuss the impact of nested queries in returning large data volumes, commenting on the impact this has both on the temporary database ('tempdb') and the total execution time.

The implementation of ORM platforms has been a mixed success, depending on the perspective of the interested observer. For application developers, the availability of an ORM to abstract away the tedious maintenance of hard-coded SQL is liberating; being able to call methods, tied to the ORM, to retrieve data is both convenient and compatible with OOPLs, fitting into the Agile, CI/CD-driven application development mindset prevalent today. However, from the perspective of the database administrator, the impact of an ORM can present serious performance issues.

The replacement of static database queries with dynamically-generated database queries has attendant problems; these include row-by-row calls (the 'N+1' problem) [15]; eager fetching causing increased I/O [10, 15]; larger execution plans, reducing the space available in the plan cache; excessive recompilations of queries suffering from lack of parameterisation; avoidance of set-theoretic constructions such as JOINs in favour of less well-performing structures like nested queries; and the avoidance of more advanced language constructions designed to facilitate efficient querying such as window functions.

In the following section we illustrate some of the issues surrounding ORMs with reference to Entity Framework, an ORM tool by Microsoft, in the setting of a web application used to manage University entrants.

IV. Demonstration

We shall use the 'Contoso University' Entity Framework example provided by Microsoft [19] against the Microsoft SQL Server 2014 RDBMS platform to illustrate selected adverse effects caused by ORM-generated queries. In the context of a University, we will 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. We will then examine the outcomes 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.

A. *Adding a student*

To begin, we 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 we manually check the table contents, we see 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)
```

B. *Getting a list of students*

In this example we have added three more students 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.

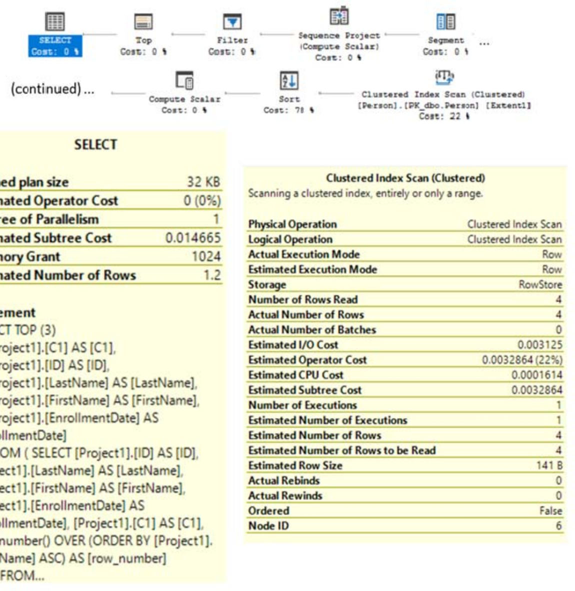
```
SELECT TOP (3)
    [Project1].[C1] AS [C1],
    [Project1].[ID] AS [ID],
    [Project1].[LastName] AS [LastName],
    [Project1].[FirstName] AS [FirstName],
    [Project1].[EnrollmentDate] AS [EnrollmentDate]
    FROM ( SELECT [Project1].[ID] AS [ID],
[Project1].[LastName] AS [LastName],
[Project1].[FirstName] 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],
            '0X0X' 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, we can write a short, and potentially more optimal SQL query:

```
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.

Let us compare the execution plans of the ORM-generated query and the query we propose, to compare the impact:



SELECT

Cached plan size	32 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	1
Estimated Subtree Cost	0.014665
Memory Grant	1024
Estimated Number of Rows	1.2

Statement

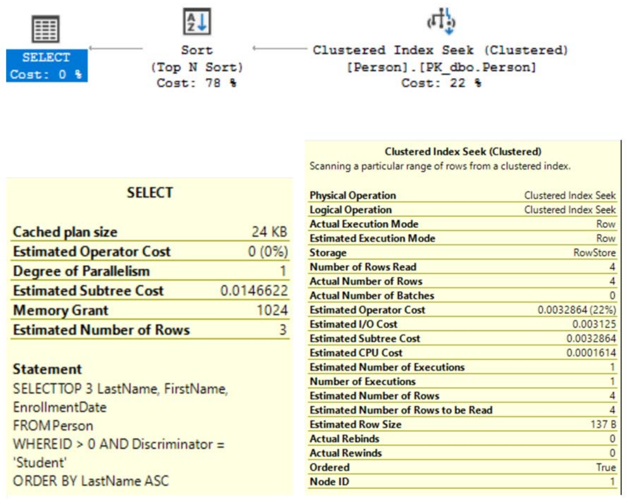
```
SELECT TOP (3)
   [Project1].[C1] AS [C1],
   [Project1].[ID] AS [ID],
   [Project1].[LastName] AS [LastName],
   [Project1].[FirstName] AS [FirstName],
   [Project1].[EnrollmentDate] AS
[EnrollmentDate]
   FROM ( SELECT [Project1].[ID] AS [ID],
[Project1].[LastName] AS [LastName],
[Project1].[FirstName] AS [FirstName],
[Project1].[EnrollmentDate] AS
[EnrollmentDate], [Project1].[C1] AS [C1],
row_number() OVER (ORDER BY [Project1].
[LastName] ASC) AS [row_number]
      FROM...
```

Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	4
Actual Number of Rows	4
Actual Number of Batches	0
Estimated I/O Cost	0.003125
Estimated Operator Cost	0.0032864 (22%)
Estimated CPU Cost	0.0001614
Estimated Subtree Cost	0.0032864
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows	4
Estimated Number of Rows to be Read	4
Estimated Row Size	141 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	6

Fig. 1. Execution plan for the ORM query for listing students



SELECT

Cached plan size	24 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	1
Estimated Subtree Cost	0.0146622
Memory Grant	1024
Estimated Number of Rows	3

Statement
SELECTTOP 3 LastName, FirstName, EnrollmentDate
FROMPerson
WHEREID > 0 AND Discriminator = 'Student'
ORDER BY LastName ASC

Clustered Index Seek (Clustered)
Scanning a particular range of rows from a clustered index.

Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	4
Actual Number of Rows	4
Actual Number of Batches	0
Estimated Operator Cost	0.0032864 (22%)
Estimated I/O Cost	0.003125
Estimated Subtree Cost	0.0032864
Estimated CPU Cost	0.0001614
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	4
Estimated Number of Rows to be Read	4
Estimated Row Size	137 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	1

Fig. 2. Execution plan for the non-ORM query for listing students

We can see 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, we may observe, in addition to our 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 [20].

C. *Editing a student*

In this example we edit a student, changing their last name and enrolment date. We note that 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)
```

D. *Searching for a student*

In this case, we have 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],
            '0X0X' 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, we can determine them to be both equal to '%Smythe%'.

We see 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 tilda. We may rewrite the SQL 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:

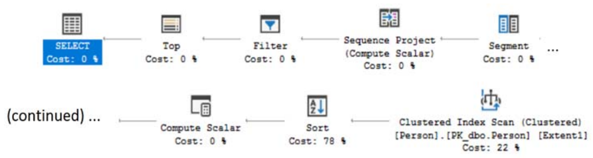


Fig. 3. Execution plan for the ORM query for student search

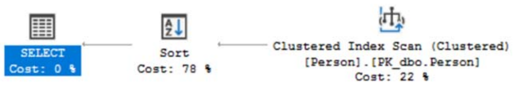


Fig. 4. Execution plan for the non-ORM query for student search

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.

E. Searching for 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)
```

We can test the performance, in terms of time and number of read operations, of each task that we have demonstrated. The results are shown below.

TABLE I. PERFORMANCE STATISTICS FOR CONTOSO SQL 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	-	-	-	-

We note 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. We also note a small 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, we may characterise the observed negative behaviour of the ORM from our demonstration as follows. We also suggest mitigations in *italics* for each characteristic but however 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)
  - *Fetch only the columns necessary for the query*
- Unnecessary nesting (subqueries)
  - *Avoid sub-querying unless necessary, use WHERE conditions or JOINS instead*
- Additional sorting (inner ORDER BY)
  - *Do not use inner ORDER BY unless also using TOP / LIMIT*
- Poor code readability (particularly aliases)
  - *Use aliases only when syntactically required*
- Poor mapping of parameters to literals (search criteria)
  - *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*

V. CONCLUSIONS AND FUTURE RESEARCH

ORM tools are widespread in today's application development environments and they are here to stay. In this paper, we have shown some of the consequences that such tools may have on query execution plan construction and query performance. Mitigations for these anti-patterns exist and tuning the ORM and database is also important.

Parameterisation can be used; eager fetching can be restricted; but some changes, such as moving from nested queries to JOINs and overcoming row-by-row selection are difficult to implement from within the ORM, leaving the database administrator to tune the database insofar as possible to accommodate this type of use.

This means that the way we think about query performance tuning in the context of ORM frameworks needs to change from a query-centric to a schema-, or database-centric approach. Future research directions may include a rethink on the construction of the cost-based optimizer, or the further integration of object-oriented principles as espoused by Date [7] into the relational model. The literature lacks a thorough review based on experimental evidence on ORM tools; future research could include the construction of a detailed ontology of observable query performance outcomes in the interests of enhancing existing ORM capabilities to improve these outcomes.

An alternative approach is the incorporation of a multi-schema model into relational database development. The declining cost and rising performance of storage means that such an approach could now be viable. A classifier can be used to redirect inbound queries sourced from an ORM to target the best-fitting schema for the query, and a potentially exciting future research direction is to incorporate this idea with the techniques available in the field of machine learning.

Unstructured databases have risen as an alternative to using relational data stores. Useful for data that does not conform to the relational model, these can also be used to work around the difficulties presented to OOPL developers working with relational databases. However, the literature contains many examples of observations where these NoSQL data stores have failed to perform as effectively nor offer the same features and benefits as the relational model. This means the continuing development of ORMs, and the ability to use meaningful performance tuning strategies, remains a relevant and important issue.

The relational model is nearly 50 years old, and from its birth as an implementation of axiomatic set theory [21] it has been augmented and expanded into international standards that span thousands of pages in documentation. Rather than abandoning the relational model for non-relational solutions, further augmentation and development of the model to incorporate the challenges posed by paradigm shifts in the way in which databases are used could prove a worthy goal.

References

[1] IBM Corporation, "10 Key Marketing Trends for 2017". Available at: https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=WRL12345USEN (Accessed 27 Mar 18), 2017.

[2] Solid IT, "DB-Engines Ranking". Available at: https://db-engines.com/en/ranking (Accessed 27 Mar 18), 2018.

[3] N. Dedić and C. Stanier, "Towards differentiating business intelligence, big data, data analytics and knowledge discovery." International Conference on Enterprise Resource Planning Systems, pp. 114-122, Nov. 2016.

[4] J.N. Cappella, "Vectors into the future of mass and interpersonal communication research: Big data, social media, and computational social science". Human Communication Research, vol. 43, issue 4, pp.545-558, 2017.

[5] M. Stonebraker, G. Held, E. Wong and P. Kreps, "The design and implementation of INGRES". ACM Transactions on Database Systems (ToDS), vol. 1, issue 3, pp.189-222, 1976.

[6] S. Balaji and M.S. Murugaiyan, "Waterfall vs. V-Model vs. Agile: A comparative study on SDLC". International Journal of Information Technology and Business Management, vol. 2, issue 1, pp. 26-30, 2012.

[7] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries and J. Kern, "Manifesto for agile software development.". Available online at http://agilemanifesto.org (Accessed May 2018), 2001.

[8] A. Miller, "A hundred days of continuous integration". AGILE '08 Conference, pp. 289-293, Aug. 2008.

[9] S.W. Ambler, Mapping objects to relational databases: What you need to know and why. Ronin International, 2000.

[10] A. Cheung, S. Maddenand A. Solar-Lezama, "Sloth: Being lazy is a virtue (when issuing database queries)". ACM Transactions on Database Systems (ToDS), vol. 41, issue 2, p.8, 2016.

[11] D. Colley and C. Stanier, "Identifying New Directions in Database Performance Tuning". Procedia Computer Science, vol. 121, Elsevier, 260-265, 2017.

[12] C.J. Date and H. Darwen, "Foundation for future database systems: The third manifesto". Massachusetts: Addison-Wesley, 2000.

[13] C. Ireland, D. Bowers, M. Newton and K. Waugh, "A classification of object-relational impedance mismatch". Advances in Databases, Knowledge, and Data Applications (DBKDA'09), pp. 36-43, Mar. 2009.

[14] B. Karwin, "SQL antipatterns: avoiding the pitfalls of database programming". Pragmatic Bookshelf, 2010.

[15] T. Chen, W. Shang, Z.M. Jiang, A.E. Hassan, M. Nasser and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping". Proceedings of the 36th International Conference on Software Engineering, pp. 1001-1012, 2014.

[16] G. Vial, "Lessons in Persisting Object Data using Object-Relational Mapping." IEEE Software, 2018.

[17] B. Jose and S. Abraham, "Exploring the merits of NoSQL: A study based on MongoDB." IEEE International Conference on Networks & Advances in Computational Technologies (NetACT), pp. 266-271, 2017.

[18] Microsoft Corporation, "Getting Started with Entity Framework 6 Code First using MVC 5." Available at: https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application (Accessed May 2018), 2009.

[19] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen and D. Wilkins, "A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective." ACM Proceedings of the 48th South East Regional Conference, pp. 42-48, 2010.

[20] C. Freedman, "Scans vs. Seeks". Available at https://blogs.msdn.microsoft.com/craigfr/2006/06/26/scans-vs-seeks/ (Accessed 29 May 2018), 2006.

[21] P. Suppes, "Axiomatic set theory". Courier Corporation, MA, 196