Query Complexity in Modern Database DSLs

ÁRPÁD GORETITY, PPCU FITB, Hungary
ISTVÁN REGULY, PPCU FITB, Hungary

ACM Reference Format:
Árpád Goretity and István Reguly. 2021. Query Complexity in Modern Database DSLs. *ACM Transactions on Information Systems* 1, 1 (July 2021), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Abstract

Domain-Specific languages have been used in databases since the 70s. Weakly-typed DSLs, such as SQL, are now joined by strongly-typed Object-Relational Mappers. There is anecdotal evidence that the use of such DSLs results in increased cognitive complexity compared to SQL, leading to tradeoffs between query correctness and productivity.

These claims are quantified, compares traditional database management systems to ORMs, and provides objective supporting evidence for the experience that ORMs have higher friction. Alternative metrics are developed for evaluating new aspects of queries.

Four widely-used, modern databases and ORMs are examined: SQLite, MongoDB, Entity Framework, and Core Data. A demo schema and queries with varying levels of sophistication are designed, representing typical business applications with diverse requirements. Complexity metrics are applied to each system and query. New metrics specific to query languages are designed and evaluated.

Query complexity is observed to be higher in ORMs than it is in traditional databases, increasing suddenly when queries not explicitly supported by the ORM are formulated. Complexity of SQL and MongoDB queries instead scales gradually with requirements, staying below that of ORM-based queries. Two new metrics, token entropy and weighted AST node count, reproduce perceived levels of cognitive complexity better than some existing measures.

1 INTRODUCTION

Maintainable, simple, and computationally efficient abstraction of databases remains an ongoing challenge. While the relational model still dominates the world of database administration, and non-relational, NoSQL systems are gaining traction as well, the proverbial impedance mismatch persists between often weakly-typed or schemaless databases and the desire for strongly-typed domain modelling approaches at the application level.

Several Object-Relational Mapping (ORM), Object-Database Mapping (ODM), and Data Mapper tools have been (and are being) developed in the industry. Most of these tools focus on the most trivial use cases, the so-called CRUD scenario (Create-Read-Update-Delete). While ORMs and similar software make it easier for developers to lift business problems from the level of raw, dynamically-typed database access libraries into the statically-typed application domain model, they can also feel limiting in the eyes of experienced programmers, while being potentially inefficient at the same time, in terms of the database access code (e.g. SQL) they generate. Accordingly, ORMs have

Authors' addresses: Árpád Goretity, PPCU FITB, Budapest, Hungary, goretity.arpad@itk.ppke.hu; István Reguly, PPCU FITB, Budapest, Hungary, reguly.istvan@itk.ppke.hu.

1046-8188/2021/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

continuously been in the focus of criticism, even having been called "The Vietnam of Computer Science" [1].

Data mapping and abstraction tools have a well-defined, narrow purpose, and as such, the raising interest in Domain-Specific Languages (DSLs) [2] shows itself in this context. Most such tools come with their own DSL, which is the primary means of interacting with the data model being abstracted over. However, this approach is not free from downsides: even moderately complex queries might require workarounds, or be impossible to write altogether, using an ORM layer. Reasons may be manifold: insufficient support for the required subset of SQL in the DSL; a desire to avoid code generation by embedding the DSL into the host language, which, however, is incapable of expressing all desired queries; or the tendency of ORMs to restrict the subset of features to a lowest common denominator, provided by all supported databases, with the goal of ensuring portability.

In this study, we examine two popular, production-ready database engines and two widely-used ORMs in order to quantitatively compare the complexity of a diverse set of typical queries, as well as the cognitive load they impose upon programmers. First, we define a schema and corresponding queries of varying levels of complexity, which work together to simulate the domain and data model for an imagined business use case. We then realize this model and the queries using each of the four software packages, using them as idiomatically as possible. Finally, we apply complexity metrics as defined in the literature to the code written for each piece of software, and we introduce our own metrics too, keeping the universal properties of query languages in mind.

Based on the obtained quantitative complexity values, we enumerate the most significant missing features and design weaknesses of each system, and explain them based on the definition and behavior of each metric. We then contrast the advantages and drawbacks of database management systems with those of ORMs, which in turn opens up the possibility for present and future authors of such systems to improve their APIs.

While computational complexity, time and space efficiency, and resource usage are paramount for the effective real-life use of database systems, they are not in scope for the present paper. Improving the execution time and memory usage of ORM-generated queries is also a topic of ongoing research, and may be considered by the authors in the future, although ease of use of high-level data abstraction systems is often at tension with the performance of data access.

2 METHODS

2.1 Databases and Data Abstraction Tools

Table 1. Databases and database abstraction layers.

Database or Framework	Version	Host Language	Version
SQLite	3	Python	3.9
MongoDB	4.4.	Python	3.9
Entity Framework Core	5.0	C#	9.0
Core Data	macOS 11.1	Swift	5.3
CoreStore	7.3.1		

The exact version of the technologies and programming languages considered is listed in Table 1. The reasons for choosing each individual database, framework, and language are elaborated below.

- SQLite [3] is a fully-featured relational database engine, widely used in embedded systems and as an application file format. It is among the most deployed databases in the world. Its approach to data storage is unique in that it works off a single regular file, and it can be linked against directly as a library, so it doesn't require setting up a separate DB server process. Consequently, it is very easy to use. The Python programming language has built-in support for SQLite.
- MongoDB [4] is one of the most well-known production-ready NoSQL databases. It is a document store, so its type system is richer than that of key-value stores. It is used extensively in the development of modern web applications, notably within the Node.JS community. Since it is inherently schemaless and dynamically typed, and it stores documents in BSON (a superset of the popular JSON data exchange format), using it from within Python is only natural.
- Entity Framework Core [5] is Microsoft's latest attempt at creating a mature, feature-complete object-relational mapper above various database management systems. It is designed primarily with the C# programming language in mind. It is also popular among web developers who prefer the .NET platform. The popular programming Q&A site, Stack Overflow, uses EF Core (among others).
- Core Data [6] is Apple's official object graph manager for macOS, iOS and other platforms in the Apple ecosystem. It supports persisting objects into an SQLite database, although it is described as "not an ORM" by its advocates. CoreStore [7] is a 3rd-party framework that builds upon Core Data, adding type safety capabilities by building upon features of the Swift programming language. Including an Apple-only framework is deemed valuable by the authors, as the company has a record of offering a unique point of view on software engineering. Furthermore, Swift is a strongly-typed language that incorporates many recent advancements in language design, being influenced by other type-centric languages such as Haskell and Rust.

2.2 The Example Schema

The example schema is designed in such a way that it makes it possible to ask different business questions about it. This results in a spectrum of queries, ranging from trivial to deeply nested ones. It also attempts to be realistic, by describing the domain model of an imaginary company that deals in lending and subletting real estate. This is a plausible scenario, as there are already several such corporations on the market.

The conceptual model of the schema is given in figure 1. Its main components are users, real estate, and bookings that connect users with real estate. In addition, login information for users is stored in the form of profiles, which represent the username and password or the 3rd-party service they authenticated with. Furthermore, sessions describe the intervals between logging in and out. Real estate are organized into a hierarchy of regions, each of which have a number of (latitude, longitude) pairs, specifying its boundary as a polygon.

We note that schema design, extraction of schema from example data, and schema mapping is a complex question and a separate field in itself. Wei and Link [8] derive a principled framework for schema design for relational databases, targeting generalizations of the normal forms 3NF and BCNF (when feasible). While we consider schema design to be of paramount importance, in this work we explicitly only target the complexity of queries and the difficulties regarding simple, effective, and usable formulation of realistically complex queries. We do not assess the complexity of schemata or the issues around efficient schema design. We also contrast the relational world with other technologies in order to provide a broader overview of the state of the art, and to thereby prove that improvement in data access and abstraction technologies is needed.

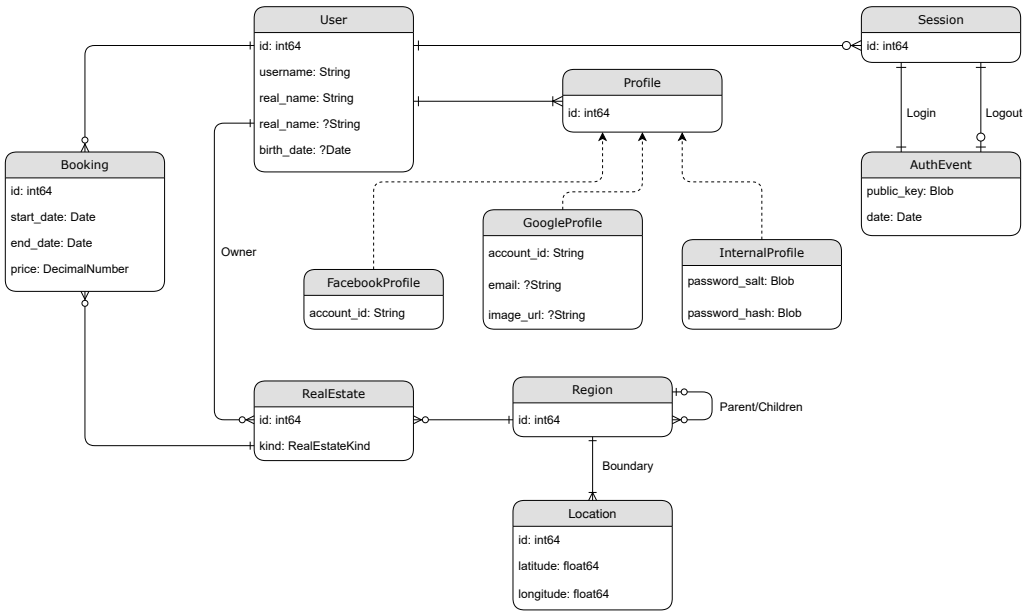

Fig. 1. E-R diagram for the example schema.

2.3 Example Queries

Each of the 11 queries exhibits a different level of complexity, and poses various kinds of challenges for database management systems. The queries try to exercise a reasonably wide spectrum of common DML features by asking questions that require a variety of techniques in order to be solved. The main goals of each query are elaborated below. For referencing the queries more concisely in subsequent sections, each query is given an ordinal number, based on their rank as reported by a specific complexity metric (token count) of the corresponding SQL implementation. This combination of metric and language is chosen as the baseline and the reference for all further comparisons, because SQL is the most widespread database query language, and code length is the simplest metric, one that most others are expected to outperform.

All implementations of the example schema, queries, and the code conducting the measurements are fully available on GitHub, at https://github.com/H2CO3/database_dsl_complexity

(1) `continents` is a very simple smoke test. It requests a list of all continents, i.e. regions with no parent region (continents are considered the top-level regions), ordered by their unique ID. It thus exercises simple selection/restriction and sorting features of the DBMS.
(2) `multi_profile_users` asks for the unique ID of users who possess more than one profile, which in turn requires grouping and aggregation, as well as filtering out groups based on an aggregated value. In the SQL implementation, this is realized by the `HAVING` clause.
(3) `no_login_users` asks for the ID of all users who never logged in, i.e. those with no associated `Session` entities at all. This requires traversing the `User` $\iff$ `Session` connection, either by means of a subquery, an explicit, `LEFT JOIN`-like operation, or by abstracting

away such an operation by means of a collection-typed `sessions` attribute on the `User` entity.

(4) `owned_real_estate_region_count` is still all about grouping and aggregation, however, it focuses on aggregation over distinct values in a group. Concretely, it returns the number of regions in which a user owns at least one piece of real estate, for each user.

(5) `num_valid_sessions` asks for the number of "valid" sessions of a user, i.e. the count of associated `Session` entities logged in but not yet logged out. This uses the grouping and groupwise aggregation capabilities of the DBMS as well, however, it is simpler than the other aggregation-centric queries, so it is also useful as a basis for comparison against other aggregation queries.

(6) `profile_counts_by_non_google_user` assesses the ability of the DBMS to deal with algebraic sum types or subtyping, by querying the count of specific variants (or subclasses, respectively) of instances of the `Profile` entity for each user. Specifically, it counts the number of Google, Facebook, and "internal" (username and password) profiles, and outputs counts of the latter two for users without any Google profiles. This requires the ability to perform pattern matching on a given variant of a profile (when using a sum type to represent it), or referring to a concrete subclass thereof (when using subtyping), as well as using it as the operand of aggregation and restriction operators.

(7) `top_n_booked_regions_for_user_x` asks the following question: for this given user, which are the most booked regions, i.e., in which regions do most of their booked real estate reside? This tests the DBMS' ability to traverse multiple connections, to sort on multiple attributes, and to limit the size of the result set.

(8) `northest_booked_latitude_slow` realizes a naïve (simple but inefficient) query for retrieving the northernmost point across all points of all regions in which a user has bookings, for each user. Therefore, it tests aggregation over multiple connections. Furthermore, the part of the data set containing coordinates for region boundaries is sized realistically (it contains more than 370 000 points), so unlike the other queries, this one is expected to run in a perceptible amount of time (taking a couple of seconds), unless it is specifically written with query performance in mind. Although the present work is not concerned with query optimization by itself, it is still useful and necessary to consider this question, because query optimization **is** needed in practice, and the fact that performance improvements often make queries more complex (as shown quantitatively in [9]) makes this problem directly relevant to our investigation.

(9) `siblings_and_parents` starts with a small, leaf region in the hierarchy of regions. It then checks the parent of said region, and obtains all siblings (i.e. all subregions with the same parent). It then recurses, repeating the same process one level higher in the tree, until root regions, i.e. continents, are reached. Such traversal of an arbitrarily-deep forest structure requires either first-class support for recursion, or an explicit language feature for graph traversal in order for the query to be simple (and efficient).

(10) `northest_booked_latitude_fast` is equivalent with its "slow" counterpart, except that it pre-computes some of the results, namely, it calculates the northernmost point of each region ahead of time, resulting in significantly faster execution. Accordingly, it demonstrates that a somewhat longer and more involved query can substantially improve performance, and thus, sometimes it is justified to make queries more complex. It also requires language support for declaring named bindings, so that the pre-computed partial results can be reused and referred to several times in the rest of the query.

(11) `avg_daily_price_by_user_by_booking_length_category` is probably the most complicated query. Conceptually, it is arranged in a number of stages. First, it computes the length

of each booking by subtracting its end date from its start date. It then classifies the bookings in five categories, based on their duration in days, such as "days" (shorter than a week), "weeks" (lasting at least a week but shorter than a month), etc. Next, for each user and each possible category, it computes the average price over all bookings, by dividing the total price of the bookings by the sum of their duration. Finally, it sorts the results lexicographically, based on the username and the textual description of the booking category. Consequently, it requires the ability to generate Cartesian products of sets, as well as multi-way conditional branching. Another detail to consider is that the resulting query will be more elegant if the data manipulation language allows the creation of inline literal sets, without the need for declaring a permant table or collection upfront.

2.4 Metrics of Complexity

For measuring the complexity of queries, proven metrics from the literature were considered. The first such measure was Cyclomatic Complexity [10]. However, it was deemed inappropriate and was subsequently ruled out, as database query languages are primarily declarative and set-based (or at least loosely collection-based), with little to no facilities for describing explicit control flow. Next, we considered the metrics introduced by Halstead in [11]. These only depend on a general notion of "operators" and "operands", and as such, they are resonably language-agnostic. They are also suitable for assessing complexity in languages without control flow instructions, furthermore, like McCabe's cyclomatic complexity, they are widely used throughout academia and industry. Therefore they are useful to compare to newly-introduced metrics. The concrete quantities are derived from four simple measurements:

N_1 = the total number of operators
η_1 = the number of *distinct* operators
N_2 = the total number of operands
η_2 = the number of *distinct* operands

from which the named metrics are computed as follows:

Vocabulary, $\eta = \eta_1 + \eta_2$
Length, $N = N_1 + N_2$
Estimated length, $\hat{N} = \eta_1 \; log_2 \; \eta_1 + \eta_2 \; log_2 \; \eta_2$
Volume, $V = N \cdot log_2(\eta)$
Difficulty, $D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$
Effort, $E = D \cdot V$

Halstead metrics represent a widely-known and frequently-used standard in measuring software complexity, and their performance has been evaluated and proven by several studies, such as [12]. Existing research shows that they are applicable even in the context of functional-declarative languages [13] [14], and in particular, to SQL [15].

Yet another class of metrics we used is a set of trivial code size indexes, namely, the number of tokens in the source code (after lexing), denoted "# Tokens", and the number of nodes in the corresponding Abstract Syntax Tree (after parsing), denoted "# Nodes". We investigated these because they are very easy to compute and they might provide interesting data in how they compare to other, more advanced metrics. They are also language-agnostic, which is an important quality when applied in a multi-lingual setting.

Finally, we introduced two of our own metrics. This was done because database complexity metrics introduced in eariler studies either considered the schema only, without evaluating queries [16], or they focused on a small number of very specific kinds of language features (such as `UNION` and `JOIN`), instead of providing a holistic view of query languages [17].

Our first contribution is Shannon entropy computed over tokens. It therefore quantifies the diversity of atomic symbols in the source text. Entropy is an information-theoretical quantity, which, when applied to a string over some alphabet, informally describes the inherent unpredictability of that string. More complicated, harder-to-understand queries are therefore expected to have higher entropy than simpler ones. It is computed using the following, widely known formula: [18]

$$S = -\sum_{i=1}^{k} p_i \, ln \, p_i$$

where the probability p_i is defined as the relative frequency of token kind i, i.e.:

$$p_i = \frac{n_i}{\sum_{j=1}^{k} n_j}$$

if there are k distinct tokens and the number of occurrences of token kind i is n_i.
So the string in our case is the list of tokens as opposed to the raw list of characters, since individual characters do not have any intrinsic meaning. For instance, a keyword or a string literal composed of many characters is conceptually not any more complicated than one that consists of a single character. Furthermore, proper tokenization allowed us to ignore whitespace and comments, which is important since these tokens do not constitute actual, executable instructions and do not contribute to the perceived complexity of the code.
The second metric we created is the number of AST nodes, weighted by the cognitive overhead of each node, mathematically:

$$W = \sum_{i=1}^{k} w_i$$

where w_i is the weight of node i. AST nodes representing trivial concepts such as literal values are assigned unit weight, and other nodes that describe more advanced operations are assigned increasingly higher weights. Weights are chosen using the following algorithm. Keywords, operators, and other tokens representing different concepts of SQL are clustered based on the frequency with which they occur in a representative subset of "favorite" queries from the Stack Exchange Data Explorer [19]. The sequence of clusters is sorted by decreasing order of frequency, and concepts or AST nodes corresponding to cluster number $i = 0, 1, \ldots$ are then assigned a weight of F_i, the i-th Fibonacci number. Fibonacci numbers are defined to start with 1 and 2 so as to ensure that clusters have strictly increasing weights.

Table 2. Frequency and weighting of a subset of T-SQL tokens in the Stack Exchange Data Explorer "Favorites" corpus, and the frequency of the corresponding SQLite constructs in our demo queries. The last column designates the specific queries in which each such token appears, using the indices under which they appear in Section 2.3.

Token/Construct	Frequency in SEDE	Weight	Frequency in Demo	Appears In Queries #
Identifier	306606	1	212	All
Numeric Literal	71803	1	7	2, 6, 11
String Literal	57378	1	10	11
AS	34130	1	25	All except 1
=	31200	2	21	All except 1, 2
SELECT	18695	2	16	All
FROM	17322	2	16	All
AND	16687	2	2	5, 11
THEN	14667	2	5	11
WHEN	14667	2	5	11
WHERE	11826	2	4	1, 3, 7, 9
COUNT	8722	2	6	2, 4, 6, 7
JOIN	8536	2	18	All except 1, 2
*	7843	2	4	1, 9
DESC	5820	2	2	7, 9
GROUP	4867	2	9	All except 1, 3, 9
OR	4793	2	1	9
INNER	4327	2	6	8, 9, 10
SUM	4220	2	3	5, 11
CASE	3918	2	1	11
IS	3660	2	5	1, 3, 5, 9
>	3588	2	1	2
NOT	3320	2	1	5
-	3264	2	6	11
UNION	2846	2	1	9
/	2736	2	1	11
TOP/LIMIT	2593	2	1	7
MAX	2283	2	3	8, 10
>=	2064	3	5	11
LEFT	1452	3	11	All except 1, 2, 7, 9
WITH	1400	3	3	9, 10, 11
HAVING	586	3	2	2, 6
VALUES	565	3	1	11
FETCH/LIMIT	479	3	1	7
CROSS	164	5	1	11
Recursion	8	8	1	9

Clustering is based on 1-D Gaussian Kernel Density Estimation in log-frequency space, and it was carried out using the SciPy stack [20]. Figure 2 shows the observed log-frequencies as a rug plot, as well as the estimated probability density function, the resulting clusters, and their assigned weight. Table 2 shows the frequency and weight of each T-SQL token of which the approximate equivalent SQLite construct appeared at least one in the example queries, along with the equivalent construct itself, and its total occurrences in the example queries.

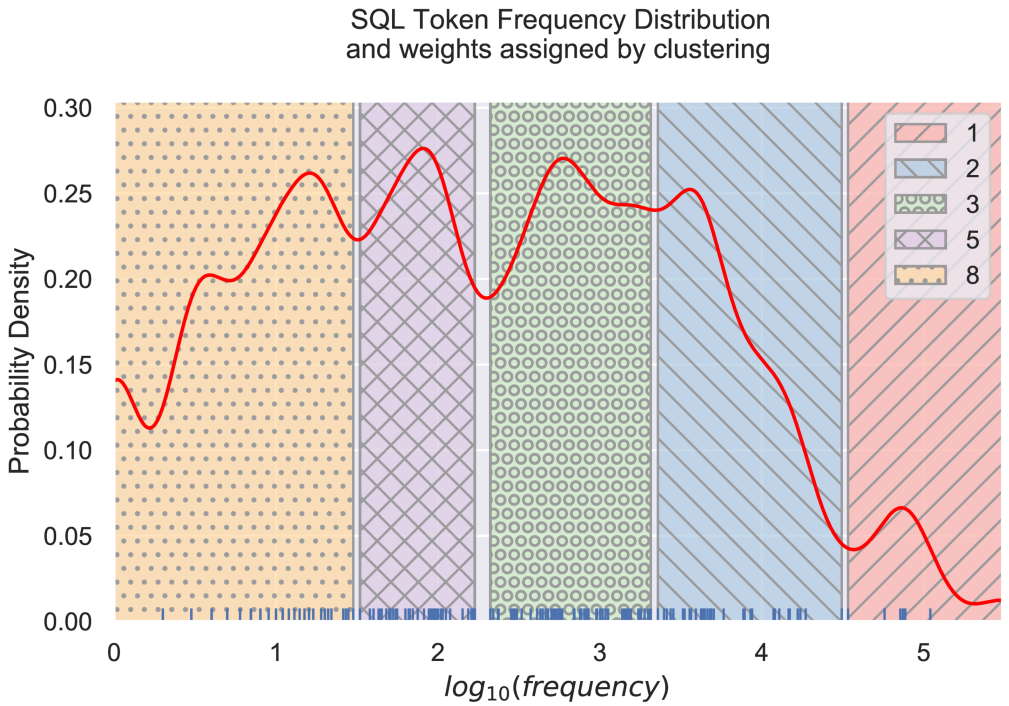

Fig. 2. Estimated probability density of T-SQL token frequency distribution, the corresponding clusters, and the assigned weights.

This scoring scheme is appropriate because Fibonacci numbers grow approximately exponentially, and humans tend to perceive ratios rather than absolute differences, so it would make little sense to assign relatively big weights which are close to each other. This line of reasoning stems from the estimation methodology of the agile approach to software development.

For query languages other than SQL, a mapping is defined between the features of SQL and those of the language being evaluated, and equivalent or approximately equivalent features of that language are assigned the same weight as that of the respective SQL constructs, ensuring that the results are comparable across different languages. This is reasonable because SQL is basically the only database query language for which it is feasible to find reasonably large, representative, and freely accessible corpora of queries. Furthermore, most working programmers are already familiar with SQL (and *only* with SQL), so they try to formulate queries for other systems based on their SQL knowledge – even if this is not the cleanest, optimally efficient, or otherwise most favorable approach.

In addition, some degree of mapping is necessary in any case, due to the variation between dialects of SQL. In particular, as already mentioned, the Stack Exchange Data Explorer corpus is written in T-SQL, the dialect for Microsoft's SQL Server, meanwhile our example queries are written in SQLite. These two systems do not use exactly the same syntax, especially when describing advanced language features. There are also keywords which do not uniquely identify an operation or language feature *per se*, e.g. the `IS`, `NOT`, and `NULL` keywords can be part of any of the expressions `IS NULL`, `IS NOT NULL`, and `NULL`. These highly ambiguous keywords have been filtered out and were not analyzed further, which is not a problem because they often correspond to trivial, atomic operations, thus they have little impact on qualitative results and consequences drawn from the measurements.

2.5 Automating Complexity Measurements

In order to make the computation of metrics feasible and objective, we have written software for each of the four languages that parses the corresponding sets of queries and computes all metrics. Where possible, we relied on existing, state-of-the-art libraries and packages for parsing source code. The following dependencies and techniques were used for each language:

- **SQL**: the `sqlparser` [21] library, written in the Rust programming language, was used for tokenizing and parsing the queries. The resulting token stream and AST nodes were postprocessed, normalized, and filtered to include only the relevant items (e.g. whitespace tokens were discarded), then the AST was traversed using a hand-written visitor pattern, and metrics were accumulated simply based on the definitions given above.
- **MongoDB**: since MongoDB queries are already specified in a structured format, no parsing was necessary in the classical sense. Lexical analysis of JSON is trivial, so it was performed using regular expressions in the host programming language, Python 3.9. Our own code then performed the postprocessing and counting similarly to the SQL case, except that a simple recursive, depth-first traversal was used, because Python being dynamically typed, there was no need for implementing a visitor.
- **C# / EF Core**: Roslyn [22] is a reusable implementation of the official C# compiler suite. It was used for parsing the module containing the target queries. There was no need for a separate lexing pass, because the resulting AST exposes individual tokens if so requested. Visitors based on Roslyn's built-in AST traversal facilities were implemented, which performed filtering of the tokens and the AST nodes, as well as the accumulation of token and node counts.
- **Swift / Core Data / CoreStore**: Libsyntax, the parsing and AST manipulation part of the official Swift compiler was used. The library was used with SwiftSyntax [23], the official but experimental Swift bindings to the low-level API, which is itself written in C and C++. The architecture is similar to that of the C# variant.

All of the code for parsing queries and computing metrics is available in the same Git repository along with the queries themselves.

2.6 Classification of Query Languages

Apart from the empirically representative nature of these languages, it is important to consider them as DSLs and therefore predict in a broader sense what advantages and downsides they bring to the table. For this kind of qualitative analysis, we use the classification introduced by Gibbons and Wu [24] and also cited by Alexandrov et al [25] for the classification of their newly-developed DSL. Based on this, the following observations can be made:

- SQL is a so-called *external* DSL. It comes with advantages such as complete design and implementation flexibility in terms of syntax, typing, and semantics, or strong coupling with

the underlying database, which results in better suitability for the problems being solved. Drawback include a lack of connection with the programming language and environment used for application development, which leads to a necessarily dyamically-typed application programming interface and a consequent lack of many compile-time correctness guarantees of queries.

- MongoDB is also an external DSL, with basically the same positive and negative properties of SQL, with the addition of some (relatively minor) differences attributed to its more uniform but repetitive syntax inherited from the JSON markup language.
- LINQ is an SQL-like construct embedded inside C# (among others). It is a so-called *deep embedding*, because LINQ expressions are abstract syntax trees which are separately JIT-compiled to SQL in Entity Framework. This means that the Entity SQL code generator has a chance to analyze, typecheck, and optimize the expression structure, however, this imposes a significant amount of work upon the implementors, and comes with a certain runtime overhead as well. When classified according to another dimension of Gibbons and Wu's system, LINQ can be considered a *quote-delimited* language, synce LINQ expressions have their own, distinguised syntax within C#, the host language.
- Core Data and CoreStore also use DSLs, embedded into Swift. In their case, the embedding is type-delimited and shallow, since Core Data employs a query and expression language based on dynamic string formatting, and its classes directly perform data manipulation and retrieval, while CoreStore provides strongly-typed query objects in some cases, but they directly translate to the dynamically-typed facilities of Core Data.

In the view of the authors of this work, an ideal new query language would need to be a deep embedded DSL with a quote-delimited high-level layer on top of a type-delimited low-level layer. These would result in several desirable properties:

- Embedding the DSL into the host language is pretty much necessary for achieving strong typing, since a generic serialization or marshalling layer needed for communication between an external database language and the general-purpose application development language would necessarily introduce dynamic typing.
- A deep embedding is preferred to a shallow one, because translating the constructs of the DSL query (written in expressions typed with types of the host language) to constructs accepted by the underlying database engine is a highly non-trivial task. Therefore, non-local analysis of the code might be useful, which means that query expressions of the EDSL are functioning as simple AST nodes (potentially carrying reified type information as well), which can be analyzed and translated to, say, SQL by a separate compiler, also provided by the EDSL library.
- The existence of a type-delimited layer is desirable because this allows the implementor of the EDSL to offload type checking to the compiler for the host language. Performing type checking correctly and efficiently, while also making sure to exactly match the semantics of the host language, would require an effort proportional to that of re-implementing the type checker of the host language, this seems like another inevitable property.
- However, extensive type-level metaprogramming is usually considered unclear, difficult to follow, or unwieldy for syntactic reasons. Thus, a quote-delimited, higher-level macro layer shall be applied on top of the type-delimited AST expressions in order to make the user interface of the DSL more palatable.

3 RESULTS

3.1 Complexity of Baseline SQL Queries

All of the aforementioned metrics have been applied to the SQL reference implementation. The results are shown in table 3, while pairwise Pearson correlations between each unique pair of metrics are given in table 4.

Metrics occupy a highly variable dynamic range. For instance, token entropy tends to fall roughly between 2 and 4, while Halstead effort easily reaches into the 10,000s. Since different metrics cannot be directly compared, and we are more interested in comparing the values of the same metric applied to different queries, we decided to make the final numeric values easier to compare and plot by normalizing them to the unit interval. This was done by translating and scaling them in such a way that for any given metric, its global minimum across all queries and databases is 0, and similarly, its global maximum is 1. Such affine transformations are permitted because they do not affect the correlation between any given pair of metrics.

The correlation between each pair of metrics is generally high. It is notable that 25 out of the 45 possible correlations exceed 0.95, meaning that such pairs of metrics are basically affine transforms of one another, containing essentially the same information with respect to the difficulty of queries. There are, however, two important exceptions: one of our own metrics, the token-based Shannon entropy, and Halstead's measure of difficulty. In fact, the correlation between token entropy and any other metric never reaches 0.9, suggesting that this metric of ours assesses the complexity somewhat differently from other constructs in the literature. Low correlations can not be attributed merely to randomness of the entropy metric, since we observe that its value does follow the perceived difficulty of the queries well – for example, it is highest for query #11, `avg_daily_price...`, and lowest for query #1, `continents`.

Table 3. Various complexity measures of the SQL reference implementation.

Metric	continents	siblings_and_parents	no_login_users	num_valid_sessions	multi_profile_users	owned_real_estate...	profile_counts...	avg_daily_price...	top_n_booked...	northest...slow	northest...fast
# Tokens	11.00	21.00	30.00	39.00	47.00	56.00	59.00	60.00	86.00	99.00	203.00
Entropy	2.40	2.76	2.61	2.88	2.95	3.01	3.23	2.97	3.05	3.15	3.39
# Nodes	7.00	13.00	13.00	16.00	20.00	25.00	29.00	26.00	49.00	45.00	113.00
Weighted	16.00	25.00	25.00	27.00	34.00	40.00	50.00	43.00	81.00	72.00	157.00
H. Vocab.	9.00	13.00	14.00	16.00	20.00	21.00	23.00	22.00	23.00	32.00	48.00
H. Length	9.00	16.00	16.00	19.00	23.00	29.00	35.00	31.00	54.00	51.00	114.00
H. Est. Len.	19.61	35.16	39.51	48.00	66.58	71.55	81.32	77.30	81.07	130.29	223.07
H. Volume	28.53	59.21	60.92	76.00	99.40	127.38	158.32	138.24	244.27	255.00	636.69
H. Difficulty	2.50	5.25	5.33	5.50	7.33	6.38	7.31	4.86	14.73	7.33	19.19
H. Effort	71.32	310.84	324.89	418.00	728.97	812.03	1156.99	671.46	3597.47	1870.00	12220.26

Table 4. Pairwise Pearson correlation between complexity metrics.

	# Tokens	Entropy	# Nodes	Weighted	H. Vocab.	H. Length	H. Est. Len.	H. Volume	H. Difficulty
Entropy	0.806								
Number of AST Nodes	0.991	0.762							
Weighted # of AST Nodes	0.988	0.776	0.997						
Halstead Vocabulary	0.983	0.864	0.955	0.953					
Halstead Length	0.993	0.787	0.999	0.999	0.963				
Halstead Estimated Length	0.987	0.830	0.965	0.960	0.997	0.969			
Halstead Volume	0.993	0.765	0.999	0.994	0.963	0.997	0.974		
Halstead Difficulty	0.893	0.717	0.924	0.935	0.825	0.923	0.825	0.906	
Halstead Effort	0.947	0.645	0.976	0.963	0.885	0.964	0.907	0.973	0.915

This seems reasonable in the light of the intuitive meaning of these measures. Most complexity metrics are influenced by the *length* of the source code, but entropy and Halstead difficulty focus on the *variation* in its constituents (tokens, operators, or operands) instead. Concretely, entropy is highest on a uniform distribution, when every token in the string occurs with equal frequency. It thus picks up many, unique or rare tokens such as keywords, and suppresses highly repetitive symbols such as parentheses or common literals like `0`. Likewise, Halstead difficulty is directly proportional to the number of *unique* operators. Of course, both of these metrics are still somewhat affected by the length of the code, but much less than e.g. weighted AST node count or Halstead program length.

3.2 Complexity across DSLs and Metrics

Accordingly, it is important to point out queries that different metrics rank differently, so as to observe the specific properties of each metric. When aggregating metrics from multiple databases and query DSLs, it becomes possible to infer patterns and trends along three different dimensions:

(1) The typical complexity of a given query, aggregated across metrics and database technologies. That is, we observe which queries are usually assigned a higher complexity by the majority of metrics, considering all tested database engines and ORMs.
(2) The typical complexity of a given database or ORM DSL. This is the complement of the previous point, that is, we observe which database technologies and languages tend to result in higher complexity values, according to all metrics and queries.
(3) Finally, we can also check how well each metric matches our subjective professional experience as well as more objective, quantitative expectations based on the nature of queries and databases. That is, we correlate metric-based ranks and complexity values with each other, or with other a priori assumptions, either over queries or databases (or both). We may thus find metrics with new, interesting properties, as well as more accurate, useful, and realistic measures of complexity. Relying on the subjective, personal experience of human programmers is deemed adequate, since it was demonstrated in [26] that opinions of professional programmers regarding the level of complexity of code tend to agree with each other to a high degree.

Our analysis is primarily concerned with issues (2) and (3) above, since the data for (1), the relative complexity of queries, is a given — the example code was intentionally constructed in such a way that queries vary from short to long, from simple to complicated. In addition, arguments (2) and

(3) are intrinsically interrelated: the goodness of metrics can only be judged based on knowledge or expectations about the behavior of databases and ORMs, as well as the feature set and level of sophistication of their respective DSLs.

In this spirit, we first quantitatively prove that different databases, ORMS, and DSLs impose clearly different amounts of churn upon the programmer, as reported by various metrics. In figure 3, Halstead effort for each query and technology is plotted. It is very apparent that the complexity of Core Data queries almost always dominates, followed by Entity Framework except in two cases. By far, the simplest queries are those written in SQL or MongoDB, without exception. Furthermore, in the case of relatively more complicated queries, complexity of MongoDB dominates that of SQL, while in the most trivial cases, they are basically on par.

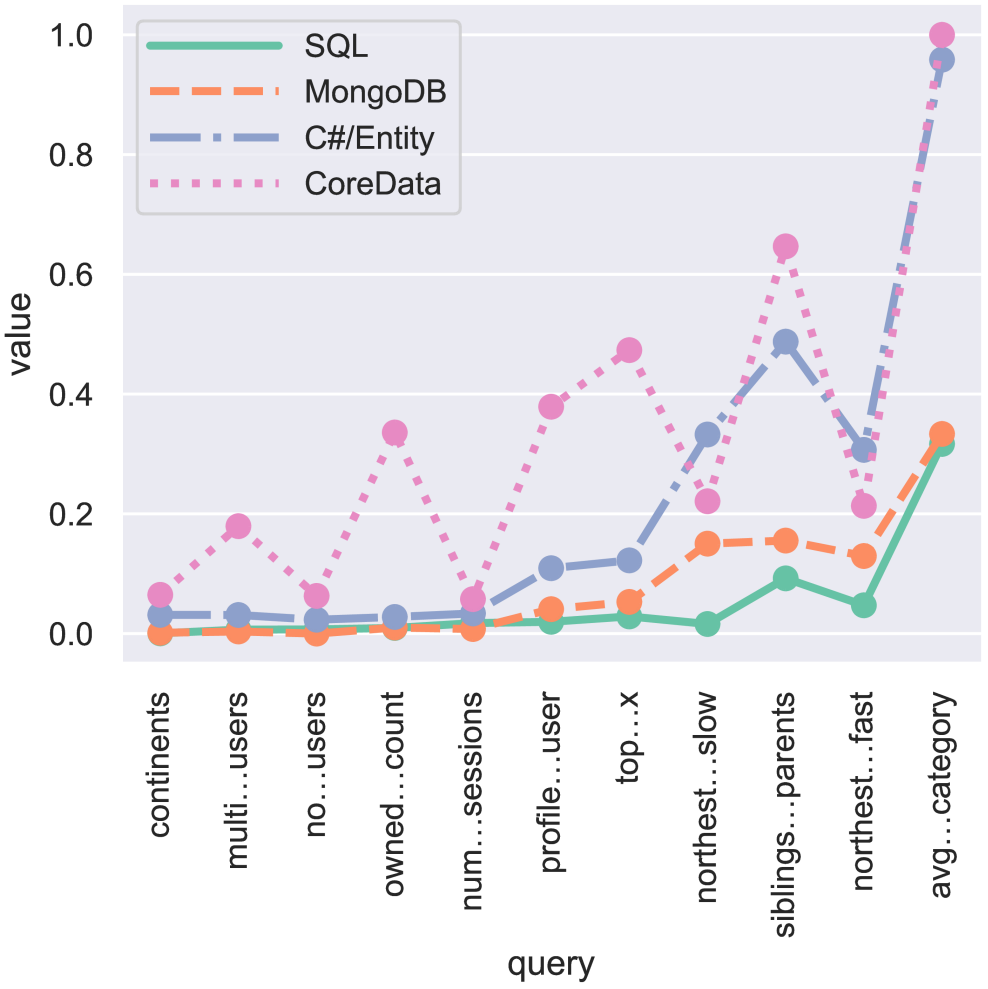

Fig. 3. Halstead Effort across databases, ORMS, and queries.

Similarly, figure 4 shows token entropy in the same arrangement. **Our token entropy metric separates technologies even more clearly,** although the ranks corresponding to SQL and MongoDB are reversed, except for the most trivial query. Comparing token entropy to the trivial, naïve, and purely length-based metric of token count (figure 5) reveals that despite having predominantly lower entropy, MongoDB queries are longer than SQL queries, with one exception, the moderately complex query #4. This is expected because the JSON-based syntax of MongoDB queries results in many repetitive and redundant tokens such as parentheses and commas, which inflate token count while simultaneously reducing entropy. Meanwhile, our weighted AST node count metric (figure 6) more accurately reflects the fact that SQL queries are generally shorter and less involved than their MongoDB equivalents.

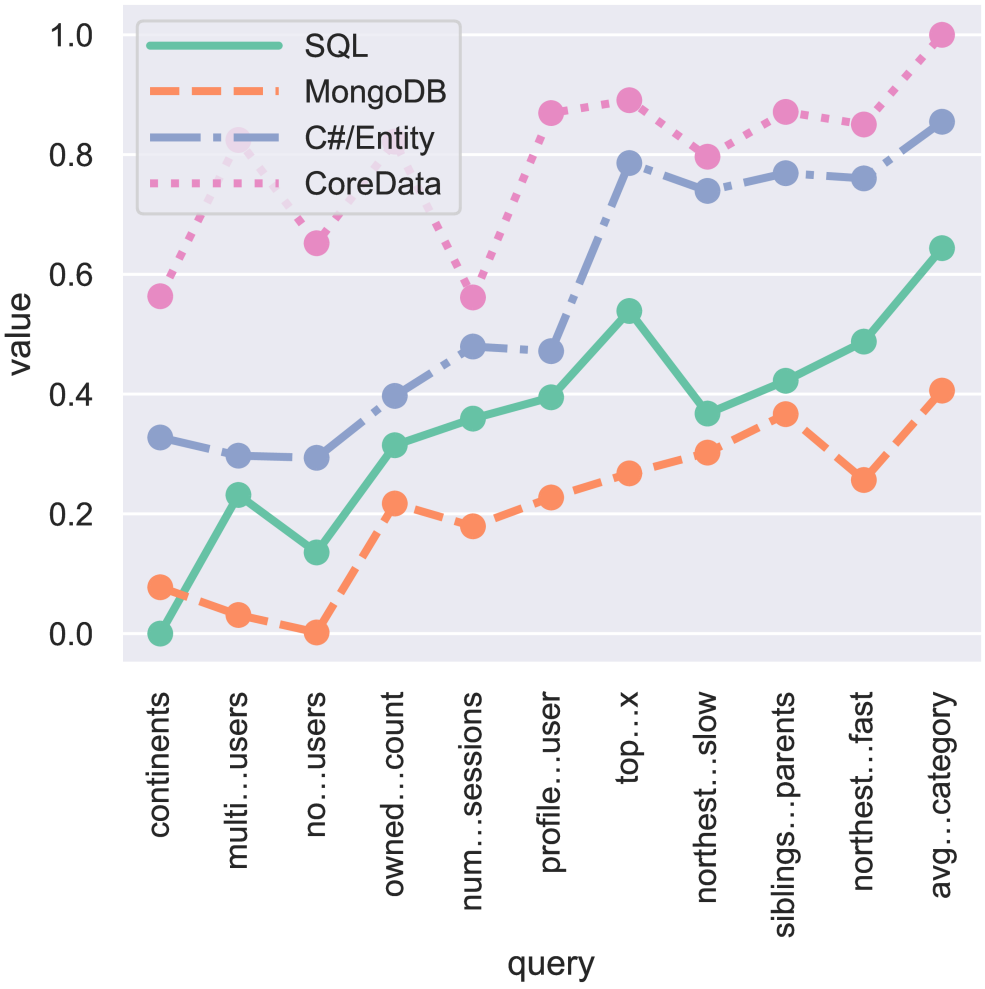

Fig. 4. Token Entropy metrics across databases, ORMS, and queries.

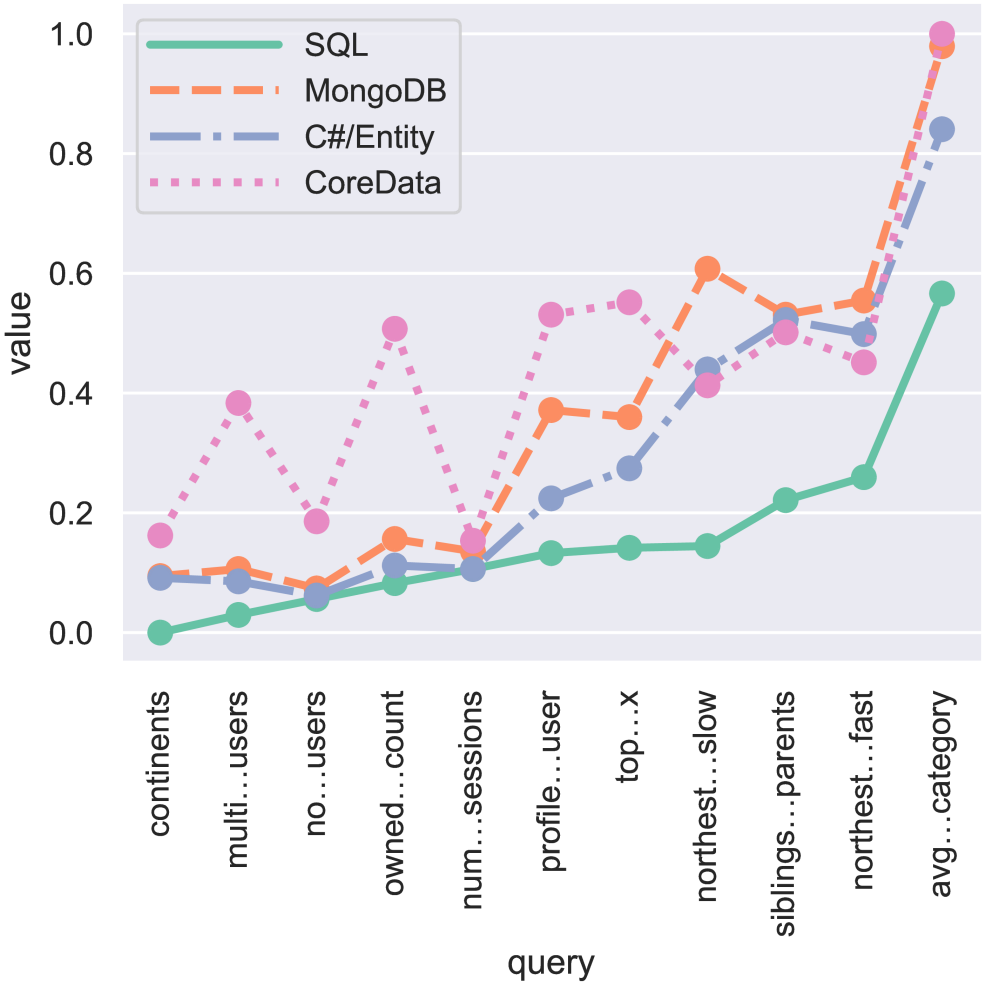

Fig. 5. Token Count metrics across databases, ORMS, and queries.

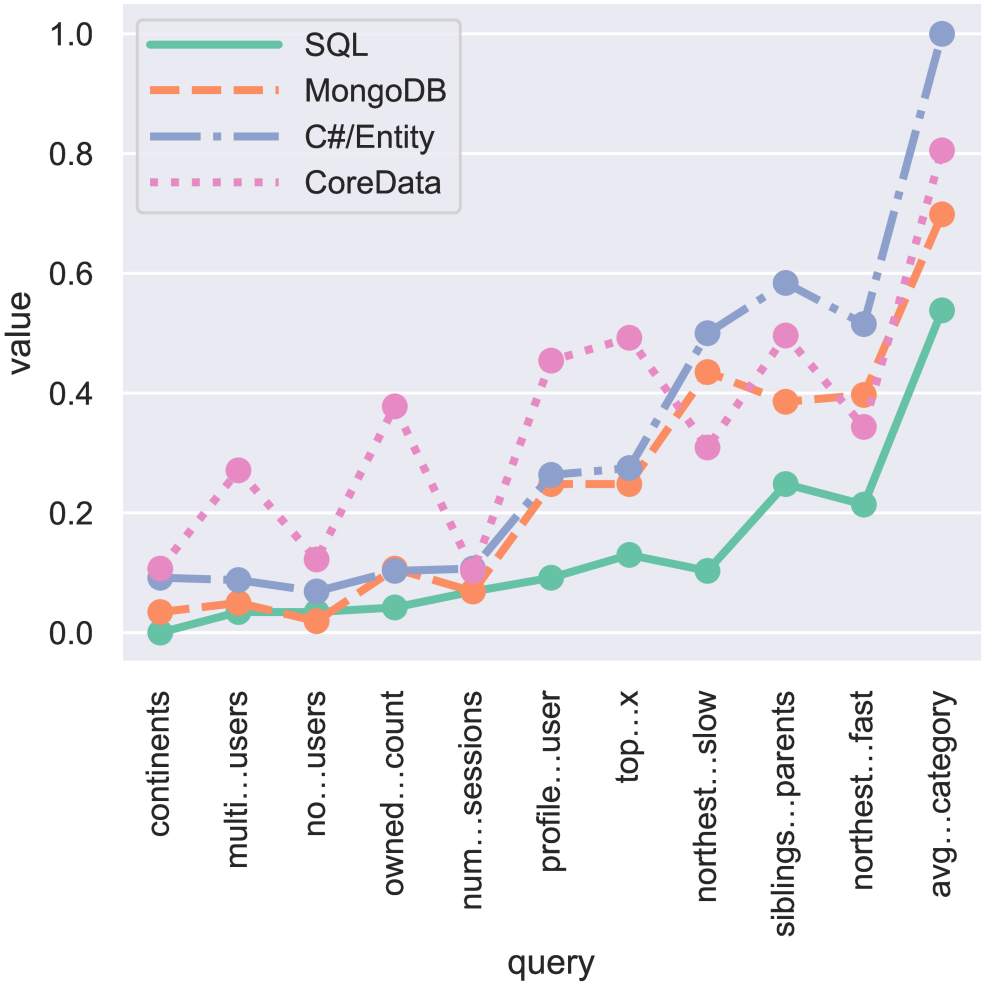

Fig. 6. Weighted AST Node Count metrics across databases, ORMS, and queries.

Next, we demonstrate that the aforementioned two contributions of ours, token entropy and weighted AST node count, provide a pair of quantities that, when considered together, capture the complexity of typical queries better than some already-existing metrics. We also show that they provide new and different insights compared to the existing metrics and to each other.

As previously pointed out, this requires observing the behavior of each database system as a function of requirements that increasingly complex queries impose. Based on professional experience, we expect that metrics of complexity, as well as perceived difficulty of the code, will increase steadily with increasing requirements in the case of low-level databases. However, we expect an abrupt increase in complexity metrics, and an overall sigmoidal-shaped complexity plot, when considering queries issued against ORMs and other higher-level data abstraction layers.

The reason behind this phenomenon is fundamental. Specialized database DSLs such as SQL and the MongoDB query language provide a diverse set of features, from the most basic to the most

advanced, with which programmers can solve problems of varying degrees of difficulty. In contrast, DSLs for ORMs focus on the few most frequently-used tasks, such as insertion, deletion, retrieval of an entire record (object) by its unique ID, or traversing a single level of a to-many connection and retrieving the associated records. However, very few ORMs provide the equivalent of features such as recursive queries, aggregations spanning multiple relations, or windowing functions. This means that once requirements for a query surpass the native feature set provided by the ORM, it becomes necessary to either build the necessary language elements from other, basic building blocks that the DSL does expose, or worse yet, to lift such more complicated parts of the query logic from the database and into the application layer. Either of these mitigations will cause a sudden increase in code complexity.

Once again, **this plateauing behavior can only be observed in the case of token entropy.** The large change in entropy is very apparent in the complexity curve of the C#/EF Core implementation, between queries #6 and #7. In the case of Core Data, this is not completely obvious at first glance, due to the ordering of the queries. However, similarly to EF Core, queries for Core Data can also be clearly classified in one of two complexity groups. Concretely, #1, #3 and #5 have a distinctly lower complexity than the rest of the queries. Incidentally, this difference is in accordance with the authors' observation that Core Data is able to compile fewer kinds of queries, and consequently, it requires adding application-level logic more frequently.

Finally, we investigate whether and why different metrics behave differently on certain queries or pairs of queries, in order to determine what purpose each metric is suited for. A particular case of interest is when the newly introduced metrics behave differently compared to Halstead's well-established formulae, and also to each other. A good example of this phenomenon is the SQL query pair #9–10 (`siblings_and_parents` and `northest_booked_latitude_fast`). Halstead difficulty and effort rank the former as significantly more complex, while token entropy reverses the relationship. Again, entropy correctly identifies the higher variation in language features being used in the latter query. This, however, does not automatically mean that it will be harder to understand for a human than the other, recursive query. This shows that a single metric does not generally capture all possible aspects of cognitive overhead, meaning that further development of new metrics is still desirable.

Another scenario worth exploring is when the apparently well-performing token entropy and the baseline token count disagree. One such example is SQL query pair #2–3 (`multi_profile_users` and `no_login_users`). Another pair is #7–8 (`top_n_booked. . .` and `northest_booked_latitude_slow`), among which the former is slightly shorter but yields higher entropy. This is in accordance with the observation that they both perform a single aggregation over a multi-way join, however, the former contains three additional language elements: a subquery, a `WHERE` restriction, and a `LIMIT` clause. Therefore, in this case, token entropy correctly identifies the cognitively heavier query. It is also worth noting that Halstead difficulty concurs with these findings. That is an important confirmatory result, because Halstead difficulty is also a measure of feature diversity (as opposed to a measure of length).

A last, important observation is that Halstead's "effort" metric works best with longer, non-trivial pieces of code. The longest and most complicated query, #11, produces almost pairwise identical values of Halstead effort for the SQL and MongoDB implementations, and for the Entity and Core Data implementations, respectively. Meanwhile, the value radically differs between ORMs and stand-alone databases, being substantially higher in the case of ORMs. This agrees with the difference between perceived difficulty of these two classes of data management systems.

4 CONCLUSIONS

Based on our analysis, it is possible to draw conclusions regarding both the applied metrics as well as as database and ORM systems.

4.1 Suitability and Fitness of Metrics

It is evident that well-established metrics such as Halstead complexity are still useful in the context of database query DSLs. In particular, Halstead effort proved useful in comparing complexity of queries of low-level, stand-alone databases to those of ORMs. However, since they do not necessarily account for unique properties of such hihgly declarative languages, further development of new, domain-specific metrics remains warranted.

Our first contribution, token entropy, proved to be the generally most robust metric, insofar as it matched real-life experience the most closely. However, it had a relatively small dynamic range of about $[2.40, 3.94]$. This means that it is primarily suitable as a tool for comparative analysis within a corpus of appropriate size. Alternatively, an effort could be made to develop an accurate absolute scale and range for this metric, making it applicable to single queries without the need for a baseline or a large corpus.

Our second contribution, weighted count of AST nodes, turned out to be a good measure of program length, as it reflects the higher-level structure of each query, in contrast with raw character or token count. Consequently, as a formatting-invariant, accurate measure of source volume, it is a useful helper for controlling for program length, thereby catching false positives reported by token entropy. So, the two metrics that we introduced work together in order to provide an accurate picture of query complexity. This is desirable, because aggregating multiple metrics increases overall accuracy even when the metrics are highly correlated [27].

In this configuration, one would inspect both token entropy and weighted AST node count. The results can then be interpreted qualitatively in the following manner:

- **High entropy and high node count** means a long query which also exhibits substantial variation in language features, therefore, it is truly and legitimately complex. For example, query #11, the aforementioned `avg_daily_price_by_user. . .` , is such, independent of the specific database software.
- **Low entropy and high node count** means a long but repetitive query. Depending on the nature of and the reading or writing tedium imposed by the repetition, this may or may not mean actual complexity. The behavior of query #6, `profile_counts_by_non_google_user`, approximates this behavior when implemented in C# / EF Core. Compared to query #5, it has more than twice the AST node count, yet its token entropy remains slightly *below* that of #5.
- **Low entropy and low node count** means a short and sweet query. Its complexity is legitimately low. An instance of such a query is #1, `continents`, when implemented in SQLite and MongoDB.
- **High entropy and low node count** cannot be observed except in extreme cases, since token entropy is bounded from above by the number of tokens: the maximal possible entropy for a string of N symbols is $\log n$, and this maximum is attained when all N symbols are distinct. Therefore, unless there are AST nodes with an unusually large number of corresponding tokens, low node count should imply low entropy. Exceptions to this rule should be evaluated on a case-by-case basis.

4.2 Issues of Current Data Abstraction APIs

The added complexity of queries against ORMs is clearly visible in almost all metrics — both in terms of length and the number of distinct language features needed for each query. This can however be justified by the increased type safety provided by ORMs in the simpler cases, as long as the use of such an ORM does not impose unreasonable restrictions upon the code. The line can usually be drawn between queries that only require natively-supported features and those that exceed this requirement, because queries in the latter category usually end up needing manual, in-memory postprocessing. This in turn increases complexity even more, while also potentially degrading runtime performance and needlessly inflating client-side memory consumption.
To summarize our findings, we enumerate the following, specific problems that we experienced and inferred from the behavior of complexity metrics.

- The friction associated with strongly, statically typed ORMS and the additional layer of abstraction between the database and the programmer is high. SQLite and MongoDB regularly lead to simpler and shorter queries compared to EF Core and Core Data. This complexity gap can be attributed to the feature sets of the respective DSLs: SQL and the MongoDB operator tree are richer and more complete, offering native support for operations such as recursion, which is hard or impossible to implement using ORMs. This in turn requires manual reimplementation of (sometimes large) parts of the query logic, resulting in longer programs overall. This is clearly reflected in structured and unstructured measures of program length. The token entropy and weighted AST node count metrics are universally higher for queries written using EF Core and Core Data, compared to their SQL counterparts.
- Relatedly, ORMs generally appear to abstract over the lowest common denominator of various kinds of databases. This may be why they usually lack support for operations other than just insertion, deletion, bulk retrieval, or single-table filtering based on very simple, atomic predicates. Specific features that our example schema would have needed, and of which the lack caused query complexity to increase:
  - Aggregations across multiple relationships.
  - Filtering based on arbitrarily complex conditions, which potentially refer to other entities and/or the result of aggregation expressions.
  - Hierarchical and/or recursive queries.
- Core Data does not provide an escape hatch from its own type system: it does not allow the programmer to drop down to SQL in order to implement unsupported operations after the fact. This is partly due to the lack of a documented, stable mapping between the high-level schema specification and the low-level storage format. The latter is a black box, although types do seem to map to tables. In contrast, EF Core supports writing raw SQL, although this is restricted to top-level queries (so-called DbSets). Both frameworks allow inspection of the machine-generated SQL, however.
- Refining the previous point, we can address the problem that ORMs today also lack the ability to extend their own DSLs by 3rd parties in a principled manner, via specialized and strongly-typed interfaces, instead of making the programmer write raw SQL by hand. These extensions would ideally be safely parameterized over the type of the backing storage engine so that, say, a recursive CTE query (which is only applicable to SQL) would not compile unless the backing database is an RDBMS.

Another important observation concerns the surface syntax of data manipulation DSLs. As evidenced by the verbose description of even simple MongoDB queries, we can conclude that JSON and BSON are raw data interchange formats, and as such, they cannot be used efficiently and ergonomically as a query language. Queries written in JSON or BSON resemble a serialized abstract

syntax tree more than a human-readable query. In contrast, SQL, which was designed to be an interactive query language right from its conception, reads more naturally and results in shorter and simpler queries. Complexity metrics confirm this observation quantitatively: while the superficially more uniform (or rather, repetitive) syntax of MongoDB results in lower values of token entropy in almost all cases except for the simplest query, it more than makes up for this in terms of query length, whether tokens or AST nodes are counted.

As far as ORMs are concerned, EF Core includes a DSL based on LINQ, which is quite powerful and convenient, at least as long as the queries remain comparatively simple. The ASTs and method calls that LINQ expressions are translated to can break down easily, failing at runtime, if the programmer writes a query of which the shape is not understood by the query generator of the framework.

This problem is even more pronounced in the case of Core Data and its wrapper, Core Store. While Core Data itself produces and accepts user-defined domain types, the query API is mostly composed of untyped or "stringly-typed" tools such as `NSFetchRequest`, `NSPredicate`, and `NSExpression`. These classes accept a DSL specified as a string literal in Objective-C or Swift, therefore most queries except for the simplest ones cannot be verified at compilation time. Meanwhile, CoreStore attempts to create a very thin, typed layer resembling an embedded DSL by the mechanism of the Swift programming language known as "key paths". This, however, breaks down once multi-table projections and aggregations are involved – for some reason, such type-safe key paths are only composable inside `Where` clauses; selecting and projecting fields across multiple classes and connections does not support chaining of typed key paths, so we need to fall back to the dynamically-typed underlying APIs of Core Data.

5 FUTURE WORK

In order to further our understanding of query complexity, it is desirable to design new metrics or improve upon existing ones, making them more accurate as well as more interpretable.

To that end, we shall address the shortcomings of the token entropy metric. To provide it with a better, increased dynamic range, it would be reasonable to compute the exponential of the entropy, e^S. This would hopefully amplify differences so that they are more easily observed. In order to attenuate the effect of repetitive, meaningless tokens, a weighted entropy metric could be introduced, in which the partial entropy of such unwanted tokens could be weighted down or outright zeroed.

Lastly, while the elucidation of problems in contemporary API design is a valuable contribution of this article, a long-term engineering challenge still remains. After having identified the issues with current ORMs and approaches to data abstraction, it would be necessary to actually design and develop a novel data abstraction framework that solves these issues. As some of the more high-level goals, such as simplicity of queries and type safety, seem to sometimes contradict each other, it is not at all obvious how such a desired API could be implemented. Further detailed design work is therefore expected to follow our investigations.

6 ACKNOWLEDGEMENTS

This work was supported by the European Union and the Hungarian government via the Human Resource Development Operational Programme (HRDOP/EFOP), grant number: `EFOP-3.6.3-VEKOP-16-2017-00`

REFERENCES

[1] Torres, A. & Galante, R. & Pimenta, M. S. & Martins, A. J. B. (2016). 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

[2] Kosar, T. & Bohra, S. & Mernik, M. (2016). Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology, 71*, 77–91. DOI: 10.1016/j.infsof.2015.11.001

[3] Hipp, R, et al. (2015). SQLite (Version 3.8.10.2).
[Software]. Available: https://www.sqlite.org/
Accessed: 16 April, 2021.

[4] The MongoDB Authors. (2021). Release Notes for MongoDB 4.4.
[Software]. Available: https://docs.mongodb.com/manual/release-notes/4.4/
Accessed: 16 April, 2021.

[5] Microsoft, Inc. (2020). What's New in EF Core 5.0.
[Software]. Available: https://docs.microsoft.com/en-us/ef/core/what-is-new/ef-core-5.0/whatsnew Accessed: 16 April, 2021.

[6] Apple, Inc. (2021). Core Data.
[Software]. Available: https://developer.apple.com/documentation/coredata Accessed: 16 April, 2021.

[7] John Estropia. (2020). CoreStore (Version 7.3.1).
[Software]. Available: https://github.com/JohnEstropia/CoreStore Accessed: 16 April, 2021.

[8] Ziheng Wei and Sebastian Link. (May 2021). Embedded Functional Dependencies and Data completeness Tailored Database Design. *ACM Transactions on Database Systems*, 46(2), Article 7. DOI: 10.1145/3450518

[9] Nurminen, J. K. (2003). Using software complexity measures to analyze algorithms — an experiment with the shortest-paths algorithms. *Computers & Operations Research 30(8)*, 1121–1134. DOI: 10.1016/S0305-0548(02)00060-6

[10] McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering, 4*, 308–320. DOI: 10.1109/tse.1976.233837

[11] Halstead, M. H. (1977). Elements of Software Science. *Amsterdam: Elsevier North-Holland, Inc.* ISBN: 0-444-00205-7

[12] Toth, Z. (2017). Applying and Evaluating Halstead's Complexity Metrics and Maintainability Index for RPG. *International Conference on Computational Science and Its Applications* DOI: 10.1007/978-3-319-62404-4_43

[13] Berg, K. & Engel, F. & Bouwhuis, D. & Bosser, T. & d'Ydewalle, G. (1992). Syntactic Complexity Metrics and the Readability of Programs in a Functional Computer Language. *Information Processing Letters*, 199-206.

[14] Nagy, C. & Vidács, L. & Ferenc, R. & Gyimóthy, T. & Kocsis, F. & Kovács, I. (2011). Complexity Measures in 4GL Environment. *2011 International Conference on Computational Science and Its Applications*, 293–309. DOI: 10.1007/978-3-642-21934-4_25

[15] Vashistha, A. & Jain, S. (2016). Measuring Query Complexity in SQLShare Workload. *Proceedings of the 2019 international conference on management of data*
[Online]. Available: https://www.semanticscholar.org/paper/Measuring-Query-Complexity-in-SQLShare-Workload-Vashistha/28ceef57eb0967f35e167d3321e62fda135f27d2

[16] Calero, C. & Piattini, M. & Genero, M. (2001). Database Complexity Metrics. *Proceedings of the 4th International Conference on the Quality of Information and Communications Technology (QUATIC), Lisboa, Portugal, March 12-14, 2001.* 79–85.
[Online]. Available: http://ceur-ws.org/Vol-1284/paper9.pdf
Accessed: 16 April, 2021.

[17] van den Brink, H. & van der Leek, R. & Visser, J. (2007). Quality Assessment for Embedded SQL. *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, Paris, France, 2007.* 163–170. DOI: 10.1109/SCAM.2007.23

[18] Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal, 27(3)*, 379–423 DOI: 10.1002/j.1538-7305.1948.tb01338.x

[19] Stack Exchange, Inc. (2021). Stack Exchange Data Explorer: Favorite Queries.
[Online]. Available: https://data.stackexchange.com/stackoverflow/queries?order_by=favorite
Accessed: 04 February, 2021.

[20] Virtanen, P. et al. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python.
Nature Methods, 17(3), 261–272. DOI: 10.1038/s41592-019-0686-2

[21] Ponomarev, N. & Grove, A. et al. (2021). Extensible SQL Lexer and Parser for Rust.
[Software]. Available: https://github.com/ballista-compute/sqlparser-rs
Accessed: 26 February, 2021.

[22] The .NET Compiler Platform authors. (2021). Roslyn: The .NET Compiler Platform.
[Software]. Available: https://github.com/dotnet/roslyn

Accessed: 08 April, 2021.
[23] Apple, Inc. (2021.) SwiftSyntax.
[Software]. Available: https://github.com/apple/swift-syntax
Accessed: 17 March, 2021.
[24] Jeremy Gibbons and Nicolas Wu. (2014). Folding domain-specific languages: Deep and shallow embeddings (functional Pearl).
Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '14). 339–347.
[25] Alexander Alexandrov, Georgi Krastev, and Volker Markl. (2019). Representations and Optimizations for Embedded Parallel Dataflow Languages.
textitACM Transactions on Database Systems, 44(1) Article 4 (January 2019), 44 pages.
[26] Katzmarski, B. & Koschke, R. (2012). Program Complexity Metrics and Programmer Opinions. *Proceedings of the 20th IEEE International Conference on Program Comprehension,* 17–26. DOI: 10.1109/ICPC.2012.6240486
[27] Yu, S. & Zhou, S. (2010). A survey on metric of software complexity. *2nd IEEE International Conference on Information Management and Engineering*, 352–356. DOI: 10.1109/ICIME.2010.5477581