Generating HQL Queries from Java Program Context

Ziyi Zhou[a], Huiqun Yu[a]*, Guisheng Fan[a], Zijie Huang[a], Kang Yang[a] and Jiayin Zhang[b]

[a]Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai, China
[b]Department of Control Science and Engineering, East China University of Science and Technology, Shanghai, China
*Corresponding Author. E-mail: yhq@ecust.edu.cn (Huiqun Yu)

Abstract

To facilitate Object-Oriented Programming (OOP) in data persistence, practitioners use a Java Object Relational-Mapping framework called Hibernate to map data bidirectionally in Java classes and data tables of Relational Database Management System (RDBMS). Specifically, Hibernate Query Language (HQL) is proposed to perform customizable queries for Hibernate in an OOP style. However, HQL queries are hard to implement and maintain due to their flexibility and complexity. To address these issues, we propose a model called HQLgen that combines deep learning and template filling to automatically generate HQL queries from program context. It employs recurrent neural network to learn the contextual information of Java program, and predicts the key elements within HQL clauses via attention mechanism. To construct the dataset for training and evaluation, we locate and extract projects containing HQL queries in GitHub followed by extensive cleaning and preprocessing, and finally obtain 24,118 HQL queries from 3,481 projects. Experimental results show that the proposed approach achieves an accuracy of 34.52% on generating simple HQL queries. In addition, we release the collected dataset for future research interest.

Keywords: Object relational mapping, HQL query generation, Deep learning, Program comprehension

Statements and Declarations

- **Competing Interests:** The authors have no competing interests to declare.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 61772200), Shanghai Natural Science Foundation (No.21ZR1416300).

1 Introduction

Software developers rely on databases to manage the retrieval and storage of data (Di Giacomo 2005). Since Object-Oriented Programming (OOP) has become a mainstream development model in the last decades, RDBMS (Relational Database Management System) such as MySQL and PostgreSQL (Di Giacomo 2005) are commonly applied since they can present relations among data tables using functionalities such as foreign keys (Cook et al. 2006). The two systems are connected by one-to-one mappings between each OOP data class and a corresponding RDMBS data table (Vial 2019). However, there still exist gaps (also known as “impedance mismatch”) (Cook et al. 2006) between RDBMS and software applications because they are completely independent systems (Vial 2019), and data persistence between them is vital for practitioners since inconsistency will harm software reliability (Meurice et al. 2016; Nazario et al. 2019). However, data persisting code could be hard to write and maintain in data-intensive software, and the difficulties usually appear in writing queries and mapping data between the software and database (Silva et al. 2019; Loli et al. 2020; Presler-Marshall et al. 2021).

Object-Relational Mapping (ORM) frameworks are designed to automatically generate and maintain database queries and data persisting code, allowing practitioners to operate database in an OOP coding style, e.g., using OOP data classes instead of tables in queries. In terms of Java applications, the most trending ORM solution is called Hibernate (Vial 2019). Meanwhile, a SQL-like query language called HQL (Vial 2019) is shipped with Hibernate to provide practitioners with the flexibility to write custom OOP-styled queries. Recent work (Presler-Marshall et al. 2021; Nagy and Cleve 2015a) reveal the difficulties and heavy cognitive load for developers to write SQL queries correctly. Compared with writing correct SQL queries, writing HQL queries is even more difficult because they are embedded in data persistence code and also incorporate the knowledge of OOP design (Vial 2019). Therefore, it is meaningful to assist developers with tools for automatic HQL generation, which could foster the correctness and efficiency of development in data persistence.

To help the majority of database users who are not proficient in query language, researchers in Natural Language Processing (NLP) design text-to-SQL (Text2SQL) generators that could translate natural languages into SQL queries (Iyer et al. 2017; Zhong et al. 2017; Xu et al. 2017; Yu et al. 2018a, c; Bogin et al. 2019; Wang et al. 2020). Meanwhile, some development frameworks and libraries also provide HQL generation functionalities based on developer-specified input. These functionalities still require the developers to write queries in advance, but in other preferred forms, e.g., type-safe DSLs (domain specific languages) of queries. However, the automatic generation of HQL queries from code context is much more difficult owning to totally different input. Specifically, the key elements in an HQL query mostly come from its program context, which involves the following two challenges. The first one is excessive large search space due to the variety of OOP specifications and code context to cover. For example, in terms of the breadth of search space, the call context of the method containing the target HQL query may be looked into for potential useful information. In terms of the depth of search space, the data classes representing the RDBMS data tables could be configured with relations, i.e., a data class may contain nested properties such as “Assignment.student.notes” where each of them is a candidate to appear in the query. The lack of precise contextual information is another major challenge. Prior Text2SQL tasks (Zhong et al. 2017; Yu et al. 2018b) use the input of instructive and detailed natural language question describing the expected behaviors of the queries. However, such input rarely exists in the program context of HQL queries. In fact, only 17% of the methods containing HQL queries are commented in our dataset, and few of their comments precisely describe the queries. Thus, effective ways to represent program context and extract information need to be designed.

To address the up-mentioned issues and provide practitioners with helpful suggestions when writing HQL queries, we propose a novel model named HQLgen to automatically generate HQL queries from Java program context. HQLgen is a data-driven approach that combines deep learning with template filling. In detail, it employs Recurrent Neural Networks (RNN) to learn the contextual information of program context, and predicts the key elements within HQL clauses via attention mechanism. To evaluate our approach, we use BOA (Dyer et al. 2015) to extract projects with HQL queries from the full mirror of GitHub by locating the createQuery method calls (Nagy et al. 2015b), and build a large dataset containing HQL queries with their related context. Our motivation is to explore the possibility of tackling this problem and develop a state-of-the-art preliminary approach, so we necessarily limit our scope to generate simple HQL queries.

The main contributions of our work are:

- We release a dataset for the task of HQL generation, which contains 24,118 HQL queries with related program context extracted from 3,481 real-world projects.
- We propose HQLgen, a model for automatically generate HQL queries from Java program context.
- Extensive experiments conducted on our dataset demonstrate the effectiveness of HQLgen, as well as the reasonableness of its input representations and model design. In particular, it achieves an accuracy of 34.52% on generating simple HQL queries.

To the best of our knowledge, this is the first work that introduces and attempts to tackle automatic ORM query generation problem.

The rest of this paper is organized as follows. In Section 2 we define our task. Section 3 summarizes the related work, while Section 4 presents how we construct the dataset. Section 5 elaborates the design of our model. We analyze the evaluation results in Section 6, and further discuss the effectiveness of our approach and threats to validity in Section 7. Finally, Section 8 concludes the paper and points out the potential future work.

2 Task Definition

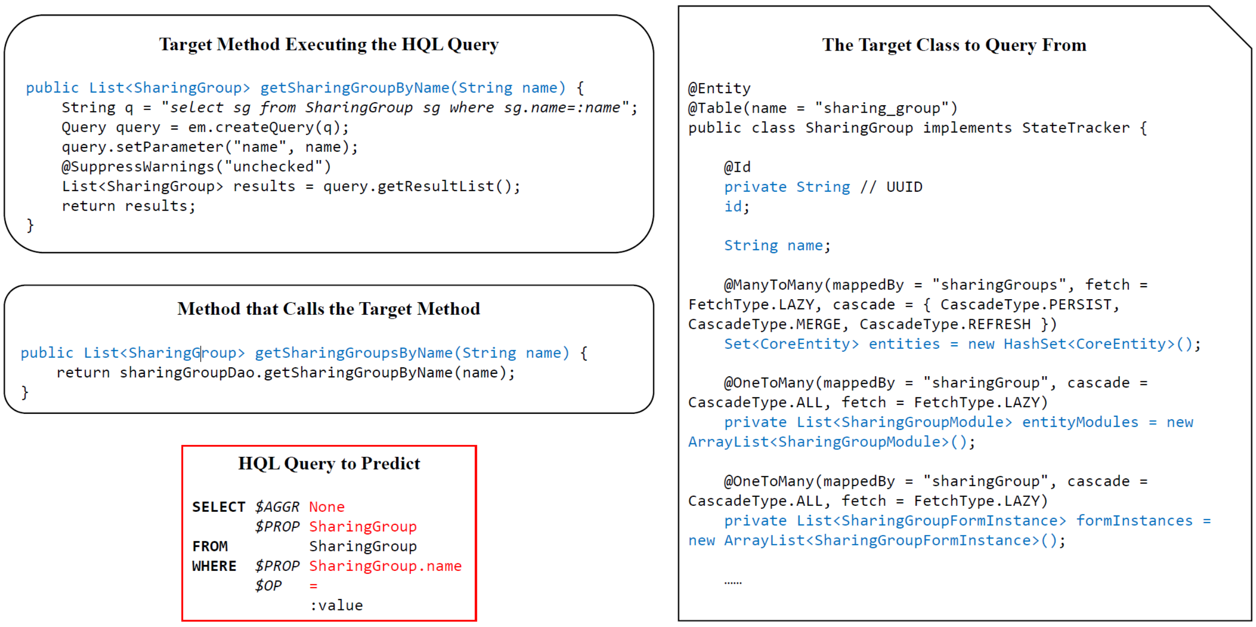
We aim to automatically generate an HQL query given its necessary program context to assist the developers. In this paper, we preliminarily focus on predicting simple queries with certain constraints. Specifically, we consider HQL queries that conform (or can be written into) the following template:

SELECT [*$AGGR*] [*$PROP*]
FROM class
WHERE [*$PROP $OP* value] [**AND** *$PROP $OP* value]*

where the slots *$AGGR*, *$PROP* and *$OP* denote aggregate function, the referred property and operator respectively. "*" means zero or more conditions concatenated with **AND**. In other words, subqueries, joins, **OR** clauses and multiple properties within the **SELECT** clause are not considered. Due to limited information presented in the context, **GROUP BY** and **ORDER BY** clauses are also not predicted. We constrain the predictable aggregate functions to {COUNT, AVG, MIN, MAX, SUM}, and the operators to {=, !=, IS NULL, IS NOT NULL, <, >, IN, NOT IN, LIKE}. Note that we treat "≤" and "≥" as "<" and ">" respectively since we observe that it is hard to tell them apart by reading the programs only. We also consider NOT LIKE the same as LIKE since they share the common purpose of vague matching, and their difference is not likely to be found in the program context. Though with these constraints, the template is able to cover most of the simple queries.

To generate an HQL query, we consider four types of context as input:

- **Target signature.** The signature of the target method that wraps the intended query.



Target Method Executing the HQL Query

```
public List<SharingGroup> getSharingGroupByName(String name) {
    String q = "select sg from SharingGroup sg where sg.name=:name";
    Query query = em.createQuery(q);
    query.setParameter("name", name);
    @SuppressWarnings("unchecked")
    List<SharingGroup> results = query.getResultList();
    return results;
}
```

Method that Calls the Target Method

```
public List<SharingGroup> getSharingGroupsByName(String name) {
    return sharingGroupDao.getSharingGroupByName(name);
}
```

HQL Query to Predict

```
SELECT $AGGR None
       $PROP SharingGroup
FROM         SharingGroup
WHERE  $PROP SharingGroup.name
       $OP   =
             :value
```

The Target Class to Query From

```
@Entity
@Table(name = "sharing_group")
public class SharingGroup implements StateTracker {

    @Id
    private String // UUID
    id;

    String name;

    @ManyToMany(mappedBy = "sharingGroups", fetch =
FetchType.LAZY, cascade = { CascadeType.PERSIST,
CascadeType.MERGE, CascadeType.REFRESH })
    Set<CoreEntity> entities = new HashSet<CoreEntity>();

    @OneToMany(mappedBy = "sharingGroup", cascade =
CascadeType.ALL, fetch = FetchType.LAZY)
    private List<SharingGroupModule> entityModules = new
ArrayList<SharingGroupModule>();

    @OneToMany(mappedBy = "sharingGroup", cascade =
CascadeType.ALL, fetch = FetchType.LAZY)
    private List<SharingGroupFormInstance> formInstances =
new ArrayList<SharingGroupFormInstance>();

    ……
```

Fig. 1 An HQL query (in italic) with its corresponding program fragments. The context we consider is marked in blue, and the elements in the query to predict are marked in red.

- **Method comment.** The comment of the target method if available.
- **Call context.** The signatures of the methods that invoke the target method if available.
- **Candidate properties.** The definitions of the properties in the target class to be queried.

All the above context can be obtained via static analysis, and complete implementations of the methods and classes are not required. The assumption here is that the target class within the **FROM** clause is known and already defined, so that the model could traverse it to search for the property to appear in the *$PROP* slot. Due to the OOP characteristics, the candidate properties involve the nested properties and the ones inherited from the parent classes.

Now we give the formal definition of our task: given the target signature, method comment (if exists), call context (if exists) and candidate properties of an HQL query, the goal is to generate the query by filling all the slots in the template above marked with "*$*". Values within the **WHERE** clauses are not predicted because we find that almost all of them are parameters while the rest are always unpredictable based on the program itself. Fig.1 shows an HQL query with its corresponding program fragments, where the context we consider is marked in blue and the elements we are going to predict are marked in red. Unfortunately, the functionality of an HQL query is seldomly explicit like that in its context, so this task is still challenging despite the constraints described above. We believe that solving the task of generating simple queries is meaningful before turning to more complicated queries.

3 Background & Related Work

3.1 ORM and HQL Analysis

Most related studies of ORM focus on the performance and complexity issues caused by the highly flexible and automated data mapping and retrieval strategies. For example, Procaccianti et al. (2016)

discover that ORM usage may increase power consumption and execution time. However, Vial (2019) argues ORM need not impact performance if used properly. Therefore, optimizers of ORM performance using different caching strategies (Chen et al. 2016a) and parameters (Singh et al. 2016) are actively developed. In terms of the quality assurance of ORM and HQL queries, Chen et al. (2016b) find these codes were frequently modified and lacked tool support. Loli et al. (2020) and Silva et al. (2019) summarize a set of technical debts regarding the sub-optimal implementations of ORM and HQL which may hinder code maintainability. As far as we know, there is no previous researches on the development process regarding the usage of ORM and HQL.

Meanwhile, in real-world engineering, there exists some HQL query generation methods. Spring Data JPA transforms method names of declared repositories (Spring components for data retrieval) to HQLs based on keyword matching[1], but requires method names written in specific formats. For instance, a method signature of “List<User> findUsersByName(String nameToMatch)” will be transformed to “select * from User u where u.name = :nameToMatch”. To reduce the error and uncertainty caused by HQLs written in plain text, QueryDSL generates data models designed for queries from ORM entity classes in advance, and facilitates the developers to write the query with IDE hints based on the generated classes[2]. For example, assuming that developers need to find all “Person” entities whose first name is Kent, they should write code such as “query.from(person).where(person.firstName.eq(“Kent”))”, where “person” is an instance of the data class generated by QueryDSL called “QPerson”, the field “firstName” is retrieved from the ORM entity “Person”, and the method “eq” is implemented by “QPerson”. Later, QueryDSL will handle the generation of ORM queries. These methods, despite that they are also generating queries, are not solving the same task as ours, and their inputs are more certain and lack complexity like contextual code.

3.2 Text2SQL

Text2SQL is a task of parsing natural language questions into SQL queries. For its great potential in human-computer interactions, it has attracted much research interest. Research on Text2SQL has a long history. Early Text2SQL interfaces focus on specific databases, and rely on domain experts to build heuristics (Warren and Pereira 1982; Androutsopoulos et al. 1993, 1995; Popescu et al. 2003). Recent approaches from database community generally involve more manual feature engineering and user interactions with the systems (Li and Jagadish 2014; Yaghmazadeh et al. 2017). Another line of studies come from the field of semantic parsing (transforming natural languages into other logical forms), which employ neural networks to translate texts into SQL queries (Zhong et al. 2017; Xu et al. 2017; Yu et al. 2018a, c; Bogin et al. 2019; Wang et al. 2020). At first, the task is considered as a sequence generation problem (Iyer et al. 2017; Zhong et al. 2017) and sequence-to-sequence model (Seq2Seq) (Sutskever et al. 2014; Cho et al. 2014) from neural machine translation is applied. Zhong et al. (2017) publish the first large-scale Text2SQL dataset named WikiSQL, which contains hand-annotated natural language questions and tables extracted from Wikipedia. Xu et al. (2017) propose a sketch-based (or template-based) approach to better leverage SQL syntax, called SQLNet. Instead of using Seq2Seq, they design a sequence-to-set approach to generate SQL queries. TypeSQL (Yu et al.

[1] https://www.javaguides.net/2018/11/spring-data-jpa-query-creation-from-method-names.html
[2] https://www.baeldung.com/querydsl-with-jpa-tutorial

Table 1 Statistics of the collected data (standard deviations are in parentheses)

Description	Statistic
Number of projects	3,481
Number of HQL queries	24,118
Number of queried classes	8,711
Number of unique tokens in the context	108,700
Number of target methods with comments	4,130
Average number of candidate properties per class	30.28 (48.67)
Average number of target method parameters	1.08 (1.23)
Average number of methods in the call context	1.89 (2.17)
Average number of conditions in the **WHERE** clause	1.06 (0.95)

2018a) further enhances SQLNet with type information of the entities in the query via knowledge graph. Concerning that WikiSQL contains only simple and single table queries, a more challenging dataset named Spider (Yu et al. 2018b) is developed. The queries in Spider include clauses like **GROUP BY**, **JOIN** and nested query, so it presents a strong challenge for current research. To address the Spider task, Yu et al. (Yu et al. 2018c) use a SQL specific syntax-tree based decoder for the generation of complex SQL queries. To better model the relations among the table schemas, several recent studies represent data tables as graph and encode them with graph neural network (Bogin et al. 2019; Wang et al. 2020). As described in Introduction, although HQL and SQL queries have similar syntax, our task is much more challenging due to complex as well as ambiguous inputs. However, many insights in Text2SQL studies are still valuable, and thus we borrow some of their model designs.

4 Dataset Preparation

4.1 Extraction of HQL Queries and Context

To build a dataset with high practicability, we extract software projects from the full GitHub 2019 October mirror using BOA (Dyer et al. 2015), i.e., an online GitHub software repository minor which is capable of mining software code ASTs at a fine-grained level. The mining script is publicly available[3]. We use the latest snapshot of the projects, and we scan if the createQuery method call presents in the project to locate potential HQL usage as (Nagy et al. 2015b) suggested, which derives 12,782 projects. Afterwards, we discard the inaccessible projects turned to private or deleted by developers, and we clone the remaining 8,799 projects from GitHub. We filter the projects that do not contain any HQL queries (e.g., the createQuery could also accept other parameters such as a Java data class object, and it will query all available data in the corresponding data table of the data class), resulting in 5,172 remaining projects. Since test and demo projects are not practical and may contain erroneous HQL queries, we also filter them out from our database by the naming patterns (e.g., names containing demo, foo, bar, example, test, or hibernate), and thus we have 3,481 available projects in our dataset.

We develop a static analysis program based on JavaParser[4] to locate HQL queries in the context

[3] http://boa.cs.iastate.edu/boa/index.php?q=boa/job/93375
[4] http://javaparser.org

(i.e., the Java class) of the createQuery method call to locate HQL queries by determining keywords such as **SELECT** and **FROM**. Note that the HQL queries constructed by concatenation of Java String and StringBuilder are manually concatenated. Then, we locate and populate the context (e.g., the methods that call the method containing the query) of the HQL queries.

4.2 Data Filtering and Preprocessing

After the extraction of textual HQL queries and their context, we use the built-in HQL structural parser of Hibernate to transfer the queries into ASTs, and uncompilable HQLs are discarded. We filter the queries according to the task definition in Section 2 using their ASTs, including:

- Discard **UPDATE** and **DELETE** queries
- Discard queries containing **JOIN**, **EXISTS** and **OR**
- Discard queries containing multiple properties within the **SELECT** clause, or more than 4 levels of data access (i.e., more than 3 ".")
- Discard queries containing customized aggregate functions
- Discard queries with tokens reflecting testing purpose, e.g., foo and bar
- Replace all value expressions with a placeholder ":value", and ensure it appears in the right of the expression, e.g., ":value <= field" will be replaced with "field >=: value"

Finally, we remove the duplicated HQL queries. As a result, we manage to locate 24,118 HQL queries with 8,711 queried classes. The dataset is released to GitHub[5], and we show its statistics in Table 1. The standard deviation of the candidate property numbers is high in that the data classes of some projects contain up to hundreds of candidate properties. To avoid our task being too simple or complicated, we further filter out the queries with only a **FROM** clause and more than 4 conditions in the **WHERE** clause. Therefore, the final size of the dataset we use is 19,228.

5 Model: HQLgen

5.1 Overall Framework

Inspired by the state-of-the-art models on Text2SQL task (Xu et al. 2017; Yu et al. 2018a), we form our task as a template filling problem. Comparing to generating the tokens in the HQL query sequentially, e.g., by applying Seq2Seq models, this approach has two advantages: it ensures the generated results are syntactically correct, and could address the unordered nature of the conditions concatenated by **AND**. When generating the **WHERE** clause, we first predict the number of conditions *#COND*, and then decide which properties will appear in this clause (since subqueries are not included, each condition relates to only one property) and assign operators to them. Similarly, the number of properties *#SEL* in the **SELECT** clause is first predicted, followed by property and aggregator selection. Fig.2 shows the overall architecture of the proposed model, HQLgen, where the arrows denote dataflows. It includes two independent predictors for the **SELECT** and **WHERE** clause respectively. The inputs will be first tokenized and embedded into real-valued vectors. Considering that an element in the code snippet can have a type and a name, we train both type and name embeddings for each token. After that, different types of context are further encoded by Long Short-Term Memory (LSTM)

[5] https://github.com/zy-zhou/HQLgen

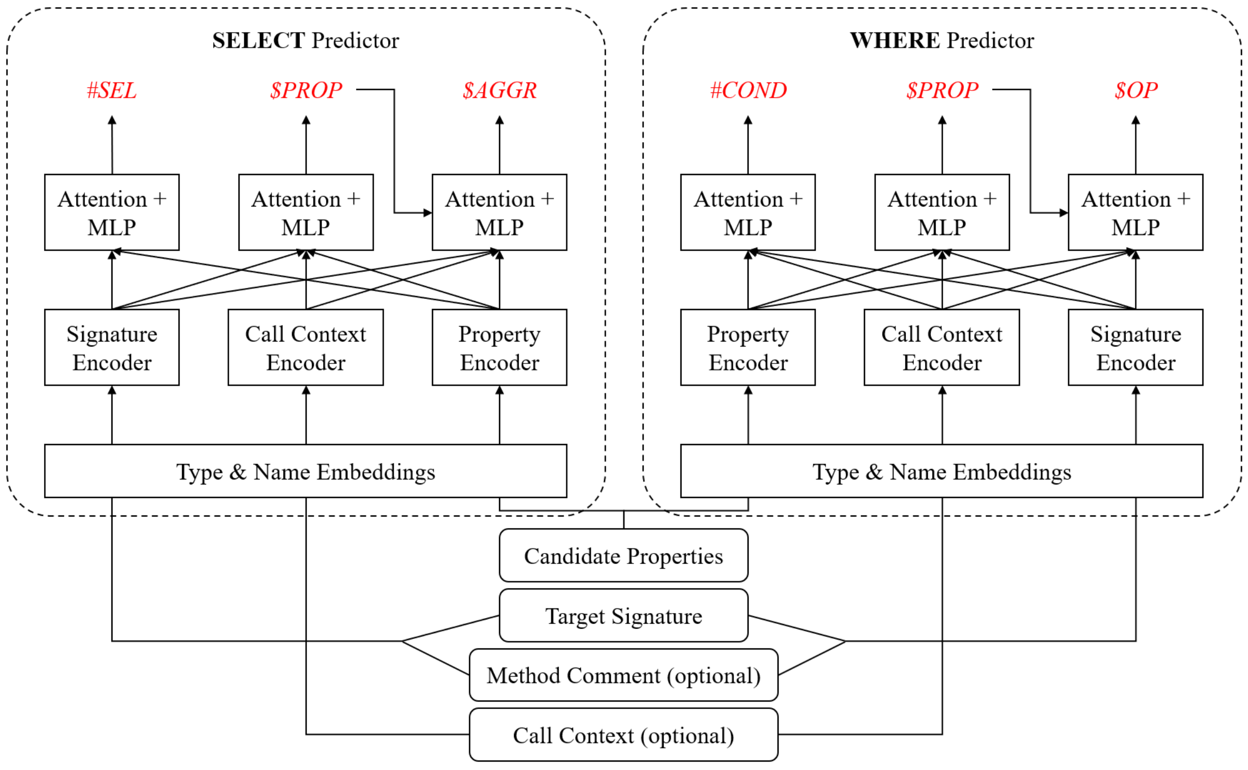


Fig. 2 Overall architecture of HQLgen. Arrows denote dataflows.

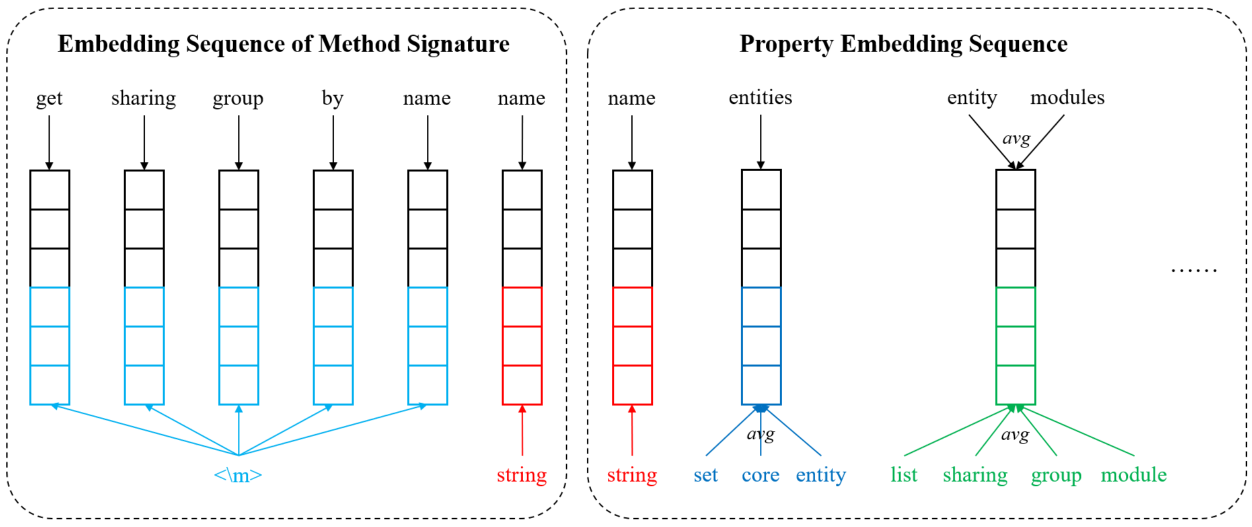


Fig. 3 Embedding sequence of method signature (left) and properties (right). Arrows denote embedding look-ups and *avg* denotes average of vectors.

(Hochreiter and Schmidhuber 1997) networks. Finally, the prediction of each slot is treated as a classification problem, in which we apply attention mechanism over the encoded context and use Multi-layer Perceptron (MLP) as classifier.

5.2 Input Representation

We first extract the context of the HQL query to be predicted via a static analysis program described in Section 4, e.g., obtain the content marked blue in Fig.1, and then encode them as follows.

5.2.1 Building Token Embeddings

Due to the naming conventions, identifiers in the program are always consist of several words or abbreviations. To better capture their semantics, each identifier in the context is first split into subtokens according to camelCase and snake_case. After that, we build a vocabulary for these subtokens, and each subtoken is mapped to an embedding vector (will be detailed later). The elements in the given context can be divided into four categories: method names, method parameters, comments and properties. They carry different amount and types of information so we treat them differently.

Parameters and properties. It is typical to transfer the parameters of an HQL query through method parameters, e.g., the parameter "name" in Fig.1. Therefore, a parameter of the target method is likely to be associated with a property that appear in the query. Based on this observation, a method parameter or property is represented by a single vector for matching purposes. In detail, its embedding is a concatenation of its name embedding and type embedding, where the name and type embeddings are the averages of their corresponding subtoken embeddings. The right part of Fig.3 illustrates the embedding vectors of the properties in Fig.1. With type information, the matching of parameters and properties will be much easier, since the ones with the same type will get similar embeddings. As described before, a property may be an instance of another class and could introduce new properties from that one, e.g., the implicit property "entities.id" in Fig.1. For such nested properties, we treat them as bags of subtokens split by "." so they can be embedded just like others.

Method names and comments. Different from method parameter, method name and comment are more informative and usually reflect the functionality of the query. Therefore, we represent a method name or comment as a sequence of its subtoken embeddings. Similarly, each embedding in the sequence is also concatenated by a type embedding. We assign a special type <\m> and <\t> to the subtokens within the method name and comment respectively, so that the model could distinguish them from the parameters and properties. The left part of Fig.3 illustrates the embedding sequence of the target method signature in Fig.1. Note that the return types are not used by HQLgen, since we found that the return values of most target methods are not query results and are noisy for predicting the query.

Accordingly, the model input consists of type subtokens (the ones within parameter and property types together with <\m> and <\t>) and name subtokens (the rest of them). They are mapped to embedding vectors through two different embedding layers:

$$\begin{aligned} \boldsymbol{e}_i^t &= \boldsymbol{W}_e^t \boldsymbol{t}_i \\ \boldsymbol{e}_j^n &= \boldsymbol{W}_e^n \boldsymbol{n}_j \end{aligned} \tag{1}$$

where $\boldsymbol{t}_i$ and $\boldsymbol{n}_j$ denote one-hot vectors of type subtoken and name subtoken, $\boldsymbol{W}_e^t$ and $\boldsymbol{W}_e^n$ are learnable embedding matrices, $\boldsymbol{e}_i^t$ and $\boldsymbol{e}_j^n$ are their embedding vectors. In this way, a same subtoken will have different embeddings if it appears both in a type or a name, e.g., the token "entity" in Fig.3, and thus it will show different semantics to the model.

5.2.2 Encoding the Context

The input context is then represented by three embedding sequences:

- The connection of target signature embedding sequence and comment embedding sequence.
- The connection of signature embedding sequences in the call context.
- Property embedding sequence.

To model the sequential information, each embedding sequence $\boldsymbol{e} = [\boldsymbol{e}_0, \boldsymbol{e}_1, \ldots, \boldsymbol{e}_{L-1}]$ is encoded by a recurrent neural network (RNN), namely bidirectional LSTM (biLSTM):

$$\boldsymbol{h}_0, \boldsymbol{h}_1, \ldots \boldsymbol{h}_{L-1} = \mathrm{biLSTM}(\boldsymbol{e}_0, \boldsymbol{e}_1, \ldots, \boldsymbol{e}_{L-1}) \tag{2}$$

where $\boldsymbol{h}_i$ denotes the hidden state of the i-th embedding and L is the sequence length. Each output element of the biLSTM, namely $\boldsymbol{h}_i$, is a semantic vector which summarizes both of the preceding elements and following elements in the sequence. Considering that the three embedding sequences have different patterns and convey different kinds of information, we apply different biLSTMs for them, as shown in Fig.2. There are two reasons for encoding the target signature with the comment by a single biLSTM. First, we find that only a few target methods are commented and such examples seem not enough to properly train a separate RNN. Second, the target signature and comment could both reflect the functionality of the target method so we expect the RNN to treat them similarly. For simplicity, we denote the encoded sequences as $\boldsymbol{h}^{target}$, $\boldsymbol{h}^{call}$ and $\boldsymbol{h}^{prop}$ respectively. Since the properties are supposed to be unordered, encoding them into a sequence seems to be questionable. However, our experimental results show that running an RNN over them could significantly improve the results, as shown later in Section 6.

5.3 Predicting HQL

5.3.1 Attention Mechanism

Inspired by SQLNet, we employ attention mechanism to let a property attend to the elements in $\boldsymbol{h}^{target}$ and $\boldsymbol{h}^{call}$. Given the hidden state $\boldsymbol{h}_i^{prop}$ of the i-th property, we compute a normalized attention weight α_{ij} for each element in $\boldsymbol{h}^{target}$:

$$\begin{aligned} w_{ij} &= \left(\boldsymbol{h}_i^{prop}\right)^T \boldsymbol{W}_a^{target} \boldsymbol{h}_j^{target} \\ \alpha_{ij} &= \frac{\exp(w_{ij})}{\sum_k \exp(w_{ik})} \end{aligned} \tag{3}$$

where $\boldsymbol{W}_a^{target}$ is a parameter matrix. Then a weighted sum of the elements in $\boldsymbol{h}^{target}$ is computed for this property:

$$\boldsymbol{h}^{target|i} = \sum_j \alpha_{ij} \boldsymbol{h}_j^{target} \tag{4}$$

Intuitively, $\boldsymbol{h}^{target|i}$ extracts the useful information in $\boldsymbol{h}^{target}$ with respect to the i-th property, and serves as a high-level vector representation of the target signature and comment (e.g., the parameter "name" will be dominating in the sequence when evaluating the property "name"). We obtain $\boldsymbol{h}^{call|i}$ similarly, but with different parameter matrix $\boldsymbol{W}_a^{call}$. Since the predictions of the slots require global information of the context, the high-level representations $\boldsymbol{h}^{target|i}$ and $\boldsymbol{h}^{call|i}$ are used as inputs of the MLP classifiers.

5.3.2 Filling the Slots

To generate the **WHERE** clause, we first predict the number of its conditions, namely *#COND*. Since we limit the max number of conditions to 4, this is cast as a 5-way classification problem with the label ranges from 0 to 4. The probability of each number is computed via an MLP:

$$\boldsymbol{p}^{\#COND} = \text{softmax}\left(\boldsymbol{W}_1^{\#C} \tanh\left(\boldsymbol{W}_2^{\#C} \sum_i \boldsymbol{h}^{target|i} + \boldsymbol{W}_3^{\#C} \sum_i \boldsymbol{h}^{call|i} + \boldsymbol{W}_4^{\#C} \boldsymbol{v}^{\#p} + \boldsymbol{b}_2^{\#C}\right) + \boldsymbol{b}_1^{\#C}\right) \quad (5)$$

where $\boldsymbol{W}_*^*$ and $\boldsymbol{b}_*^*$ denote parameter matrices and bias vectors (same notations will be used below) and $\boldsymbol{v}^{\#p}$ is the one-hot vector of the number of parameters in the target method. As mentioned before, a parameter of the target method is possibly related to a property in the **WHERE** clause, so we consider $\boldsymbol{v}^{\#p}$ as an important feature to predict *#COND*. Next step is to compute the probability of the *i*-th property to appear in the **WHERE** clause:

$$p_i^{\$PROP\_C} = \sigma\left((\boldsymbol{u}^C)^T \text{ReLU}\left(\boldsymbol{W}_1^C \boldsymbol{h}_i^{prop} + \boldsymbol{W}_2^C \boldsymbol{h}^{target|i} + \boldsymbol{W}_3^C \boldsymbol{h}^{call|i} + \boldsymbol{b}_2^C\right) + \boldsymbol{b}_1^C\right) \quad (6)$$

where $\boldsymbol{u}^*$ denotes a parameter vector (same notations will be used below), δ denotes sigmoid function and ReLU denotes the rectified linear unit (Nair and Hinton 2010). To avoid ambiguity, we rename the *$PROP* slot in the **WHERE** and **SELECT** clause to *$PROP_C* and *$PROP_S* respectively. During prediction, we select *#COND* properties with the highest probabilities to form the **WHERE** clause. For each of these properties, we assign it an operator through a 9-way classification (available operators are described Section 2):

$$\boldsymbol{p}^{\$OP|i} = \text{softmax}\left(\boldsymbol{W}_1^O \tanh\left(\boldsymbol{W}_2^O \boldsymbol{h}_i^{prop} + \boldsymbol{W}_3^O \boldsymbol{h}^{target|i} + \boldsymbol{W}_4^O \boldsymbol{h}^{call|i} + \boldsymbol{b}_2^O\right) + \boldsymbol{b}_1^O\right) \quad (7)$$

where $\boldsymbol{p}^{\$OP|i}$ contains the probabilities to choose each operator for the *i*-th property.

Since we limit the **SELECT** clause to include no more than one property, the value of *#SEL* can only be 0 or 1. Different from predicting *#COND*, the call context is not used when predicting *#SEL* because we find this yields better results. Therefore, the probability of *#SEL* = 1 is computed as

$$p^{\#SEL} = \sigma\left((\boldsymbol{u}^{\#S})^T \tanh\left(\boldsymbol{W}_1^{\#S} \sum_i \boldsymbol{h}^{target|i} + \boldsymbol{b}_2^{\#S}\right) + \boldsymbol{b}_1^{\#S}\right) \quad (8)$$

If *#SEL* = 1, the choosing of *$PROP_S* is formed as multi-class classification:

$$\boldsymbol{p}^{\$PROP\_S} = \text{softmax}\left(\boldsymbol{W}_1^S \text{ReLU}\left(\boldsymbol{W}_2^S \boldsymbol{h}_i^{prop} + \boldsymbol{W}_3^S \boldsymbol{h}^{target|i} + \boldsymbol{W}_4^S \boldsymbol{h}^{call|i} + \boldsymbol{b}_2^S\right) + \boldsymbol{b}_1^S\right) \quad (9)$$

Finally, for the aggregator, we compute

$$\boldsymbol{p}^{\$AGGR} = \text{softmax}\left(\boldsymbol{W}_1^A \tanh\left(\boldsymbol{W}_2^A \boldsymbol{h}_i^{prop} + \boldsymbol{W}_3^A \boldsymbol{h}^{target|i} + \boldsymbol{W}_4^A \boldsymbol{h}^{call|i} + \boldsymbol{b}_2^A\right) + \boldsymbol{b}_1^A\right) \quad (10)$$

where the output probabilities are for the 5 aggregators mentioned in Section 2 plus "no aggregator". If *#SEL* = 0, the term $\boldsymbol{W}_2^A \boldsymbol{h}_i^{prop}$ is omitted from (10). In this case, the aggregation is performed over the objects of the target class.

During inference, the slot values with the highest probabilities computed by HQLgen are selected.

After that, we recover the whole HQL query according to the predicted values and the template defined in Section 2.

5.4 Training

The prediction of *#COND*, *$OP*, *$PROP_S* and *$AGGR* are multi-class classification problems, so they use standard cross-entropy loss:

$$\mathcal{L}^{*} = -\sum_{i=0}^{N-1} y_i \log(p_i^{*}) \tag{11}$$

where N is the number of classes, $y_i = 1$ if the label is i else 0, and "*" denotes one of the above slots. The prediction of *#SEL* is a binary classification problem, so its loss function is

$$\mathcal{L}^{\#SEL} = -[y \cdot \log(p^{\#SEL}) + (1-y)\log(1-p^{\#SEL})] \tag{12}$$

where y is the true value of *#SEL*. The prediction of the properties in the **WHERE** clause is similar to a multi-label classification, but the total number of labels (i.e., the number of candidate properties) is varied for each example. Therefore, it uses binary cross-entropy loss:

$$\mathcal{L}^{\$PROP\_C} = -\left[\sum_{i=0}^{P-1}\left(\alpha y_i \log\left(p_i^{\$PROP\_C}\right) + (1-y_i)\log\left(1-p_i^{\$PROP\_C}\right)\right)\right] \tag{13}$$

where P is the number of candidate properties for the given HQL query, $y_i = 1$ indicates that the i-th property actually appear in the **WHERE** clause and $y_i = 0$ otherwise. Here the hyperparameter α is set to 3 following Xu et al. (2017). Because most of the candidate properties do not occur in the **WHERE** clause, α is introduced to reward the model if it chooses a correct property.

As shown in Fig.2, HQLgen includes two separate modules: a **SELECT** predictor and a **WHERE** predictor. Their loss functions $\mathcal{L}^{SEL}$ and $\mathcal{L}^{COND}$ are computed by

$$\mathcal{L}^{SEL} = \mathcal{L}^{\#SEL} + \mathcal{L}^{\$PROP\_S} + \mathcal{L}^{\$AGGR} \tag{14}$$

$$\mathcal{L}^{COND} = \mathcal{L}^{\#COND} + \mathcal{L}^{\$PROP\_C} + \mathcal{L}^{\$OP} \tag{15}$$

Since the prediction of *$OP* and *$AGGR* is depend on the results of *$PROP_C* and *$PROP_S* respectively, we pass the ground truth values of the latter to (7) and (10) during training, so that these submodules can be better guided. The learnable parameters (including the embedding layers, biLSTM encoders and MLPs) of the two predictors are not shared, which means they can be separately optimized. We also tried to use shared embedding layers for them but it shows slightly worse performance.

5.5 Implementation Details

The dimensions of name and type embeddings are both 96, while the hidden size of the encoders is set to 32. We initialize the parameters using Glorot initialization (Glorot and Bengio 2010). The training objectives (14) and (15) are optimized using Adam (Kingma and Ba 2014) with initial learning rate of 0.001 and batch size of 32. We clip gradient norm by 5 and apply dropout of 0.4 on the embedding vectors and outputs of the encoder LSTMs. To avoid over-fitting, we apply early stop to both predictors.

Specifically, we stop training when the validating accuracy (will be explained in Section 6) of the clauses do not improve within 4 epochs and pick the best model. Both predictors are trained for no more than 25 epochs. We implement HQLgen using PyTorch 1.7 with CUDA 11.0 and CuDNN 8.0.5. It is evaluated on a CentOS server with an RTX3090 GPU.

6 Evaluation

6.1 Evaluation Settings

We use matching accuracy to evaluate the queries generated by HQLgen quantitively. In detail, we rewrite the ground truth HQL queries according to the template defined in Section 2, and then examine whether they match the generated ones ignoring the order of the **WHERE** conditions. After that, we compute the following metrics:

- **ACC.** Proportion of the generated queries that fully match the ground truth queries.
- **ACCSEL.** Proportion of the correctly generated **SELECT** clauses. This is the stop criterion of the **SELECT** predictor.
- **ACCWHERE.** Proportion of the correctly generated **WHERE** clauses. This is the stop criterion of the **WHERE** predictor.
- **ACCPROP_S.** Accuracy of the predictions of *$PROP_S*. *#SEL* = 0 is considered as an extra label for *$PROP_S* when computing the accuracy here.
- **ACCAGGR.** Accuracy of the predictions of *$AGGR*.
- **ACC#COND.** Accuracy of the predictions of *#COND*.
- **MCOND.** Proportion of the generated **WHERE** clauses that have at least one correct condition.

These metrics reflect the overall performance of HQLgen and its submodules. A better metric could be the execution accuracy of the query results. Unfortunately, it is impossible to compile all the collected projects and execute the queries. We will try to calculate the execution accuracy on some high-quality projects in the future.

We employ 5-fold cross validation to obtain predicted results of all examples. When splitting the data, we use two different strategies: mixed and cross-project. With the first setting, all examples are randomly divided into 5 folds. With cross-project setting, the examples from the same project will be assigned to a fold, so that the projects in the test set are not exposed during training and the generalization ability of HQLgen is further evaluated.

6.2 A Rule-based Baseline

To the best of our knowledge, there is no effort on automatically generating HQL queries from program context. Nevertheless, to better verify the effectiveness of HQLgen, we develop a rule-based model as a baseline to compare with. It is built upon massive observations and statistical characteristics (will be detailed in Section 6.4) of the code corpus. The overall objective of this baseline is same as HQLgen, i.e., to fill the template defined in Section 2, but the predictions it makes is based on various rules described below.

Prediction of *$AGGR* and *#SEL*. We first check whether the target method name contains any aggregator name. If true, *$AGGR* will be filled with the occurred aggregator, else the baseline considers

Table 2 Accuracy of the generated HQL queries

	Approach	ACC	ACC$_{SEL}$	ACC$_{PROP\_S}$	ACC$_{AGGR}$	ACC$_{WHERE}$	ACC$_{\#COND}$	M$_{COND}$
	Rule-based Baseline	19.19%	75.44%	79.71%	92.44%	32.88%	38.46%	30.62%
Mixed	HQLgen	**34.52%**	**81.78%**	**84.09%**	**94.79%**	**39.98%**	71.16%	50.97%
	-call context	34.04%	80.75%	83.07%	94.60%	39.46%	**71.69%**	**51.51%**
	-type embedding	29.16%	74.81%	77.05%	94.37%	35.37%	68.41%	49.18%
	-property encoder	29.30%	76.22%	78.59%	93.20%	35.20%	70.32%	45.59%
	-target parameter	25.75%	77.24%	79.82%	94.22%	30.60%	63.40%	44.88%
Cross-Project	HQLgen	**29.11%**	75.83%	78.44%	93.19%	**34.50%**	**68.09%**	**44.52%**
	-call context	28.85%	**76.88%**	**79.51%**	93.27%	34.27%	67.81%	43.71%
	-type embedding	24.52%	73.09%	75.72%	93.43%	29.98%	64.30%	41.18%
	-property encoder	24.69%	72.12%	74.88%	92.49%	30.26%	68.46%	38.88%
	-target parameter	20.90%	72.77%	75.22%	**93.94%**	25.41%	62.20%	35.80%

that no aggregator is used in the **SELECT** clause. For example, given a target method name "getCount", the baseline will choose COUNT as the aggregate function. Since we find that most of the queries in our dataset do not select a specific property, the baseline simply assigns 0 to *#SEL*.

Prediction of *$PROP_C* and *$OP*. As stated before, the target method parameters are likely to be associated with the properties in the **WHERE** clause. Therefore, for each target method parameter, we check whether it matches a candidate property. If true, a **WHERE** condition will be generated for the matched property. Here we consider two variable matches if both of their tokenized version of type and value (split according to Section 5.2.1) matches. For example, a parameter "int categoryID" matches a property "int category_id". Since "=" is the most frequently used operator in the conditions, the baseline simply assigns "=" to *$OP*.

Since we focus on the predictions of simple queries, we assume that such a well-designed heuristic model could be a strong baseline against HQLgen. In fact, queries like the ones in Fig.1 can be easily generated by this baseline. Besides, it should be noted that this baseline is an unsupervised approach so it will not be affected by the data splitting strategy.

6.3 Accuracy of the Generated Queries

In this section, we report the scores in terms of above metrics achieved by HQLgen and the baseline. We also conduct an ablation study to examine the effectiveness of the mechanisms proposed in Section 5, by introducing the following variants of HQLgen:

- ***-call context.*** The call context is not considered by HQLgen.
- ***-type embedding.*** Type embeddings are not learned for the context. Instead, only name embeddings are built and fed into the encoders.
- ***-property encoder.*** Replace the biLSTM encoder for the property embedding sequence by a fully-connected neural layer with tanh activation. This encodes the embeddings into vectors of 64-dementional, which is the same dimension as the biLSTM outputs.
- ***-target parameter.*** Remove the method parameters from the target signature.

Their results are shown in Table 2. As seen, the baseline could accurately generate 19.19% of the queries, denoting that the rules we defined are effective. Compared to the baseline, HQLgen achieves significantly higher accuracy on full query match. Specifically, it achieves 34.52% and 29.11% ACC scores, which are acceptable scores considering the limited information provided by the input context. HQLgen also outperforms the baseline on all metrics except for $\text{ACC}_{\text{PROP\_S}}$ under different data splitting strategies. Checking the scores obtained by the two predictors, it is clear that the **WHERE** clause is much more challenging to predict. The number of correctly generated **WHERE** clause is less than half of the correct **SELECT** clause for both of the baseline and HQLgen. We can find that *$AGGR* is the easiest to predict, since most of the queries do not include an aggregate function. Most of the queries do not select specific properties as well, so $\text{ACC}_{\text{PROP\_S}}$ is also high. HQLgen performs obviously better on predicting the **WHERE** clause than the baseline. HQLgen achieves more than 44% and 50% on M_{COND} with different settings, i.e., almost half of its predictions capture at least one correct condition, while the baseline only achieves 30.62%. However, its $\text{ACC}_{\text{WHERE}}$ scores are around 10% lower than M_{COND}. This suggests that about 10% of the predicted queries are flawed due to missing conditions in the **WHERE** clause. Due to the lack of detailed knowledge of the target class and its properties, some conditions cannot be predicted from the context we use. Overall, HQLgen is proved to be effective compared with the baseline. The main reason is that the neural model we designed can better handle the semantic associations between the identifiers in the program, while the heuristics are heavily dependent on strictly named identifiers and are not suitable in many situations. We will further discuss this in Section 7. An interesting phenomenon is that although the ACC_{SEL} and $\text{ACC}_{\text{WHERE}}$ scores of HQLgen on cross-project setting are close to the baseline, it still obtains much higher ACC score. This suggests that their predictions are orthogonal on some examples. In future work, it is worth trying to enhance HQLgen via heuristics.

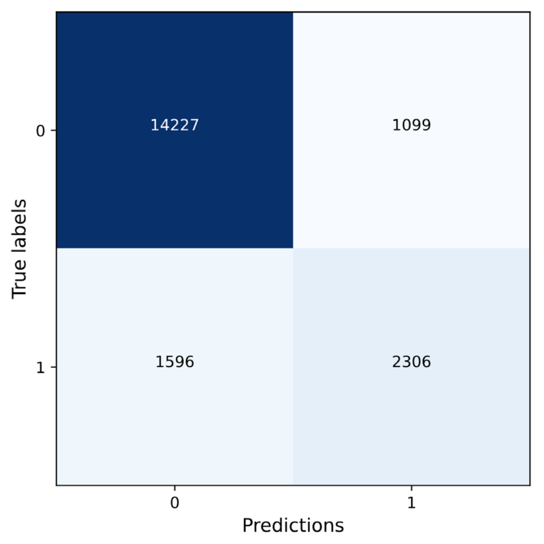
Comparing HQLgen to its variants, it shows the highest ACC scores on both splitting settings, which demonstrates the effectiveness of our designs. However, the performances of its submodules are not always the best. With cross-project setting, *-call context* scores higher on the **SELECT** clause, indicating that the call context is less useful to predict this clause when generalizing to new projects. When the projects are mixed, *-call context* scores higher on $\text{ACC}_{\text{\#COND}}$ and M_{COND}. Since the call context may not highly related to the target HQL query, ignoring it could make the outputs of the submodules less biased in some cases. However, the call context is still proved to be helpful considering the overall performance, especially the match of the full **WHERE** clause. Removing the type embeddings results in obviously lower results on all metrics, which demonstrates that the types are extremely useful for matching the parameters and properties, and our input representations are effective. Replacing the RNN encoder for the property sequence by a nonlinear layer leads to significantly worse scores as well. Although the properties are supposed to be unordered, encoding them together in a sequence could provide a global view of the target class, and thus the relationship between the properties can be better modeled. Ignoring the target method parameters results in even bigger decrease in ACC, which verifies our judgement on their importance in choosing the properties. Despite that most scores of *-target parameter* are worse than HQLgen, its ACC_{AGGR} is better with cross-project setting. We assume the reason could be that the aggregator is easier to be inferred from only the method name when there is no knowledge from the same project. Since the code within different projects may highly varied due to different domains, programming styles and vocabularies, the cross-project performance of HQLgen is relatively low, denoting that mining useful context within projects is important.

6.4 Analysis on *#SEL*, *$AGGR*, *#COND* and *$OP*

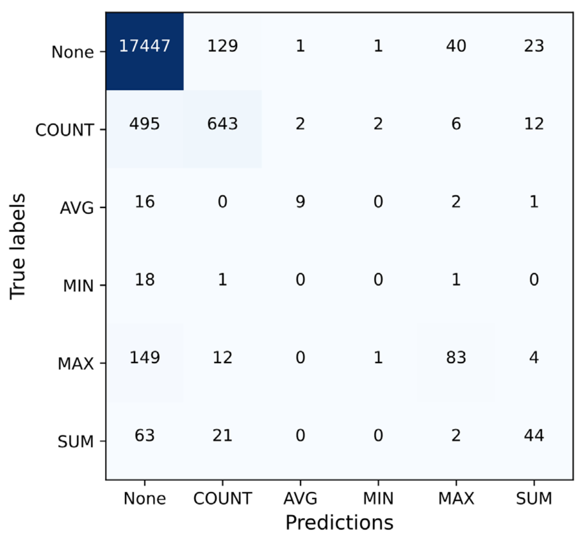
The predictions of *#SEL*, *$AGGR*, *#COND* and *$OP* are formed as multi-classification problems. In this section, we further analyze the model performance on these slots by examining the confusion matrices and F-measure scores, as shown in Fig.4 and Table 3 respectively. Specifically, to take label imbalance into account, we compute the weighted macro-F1 scores:

$$F1_i = \frac{2 \cdot P_i R_i}{P_i + R_i}$$
$$macro\text{-}F1 = \sum_i w_i F1_i \tag{16}$$

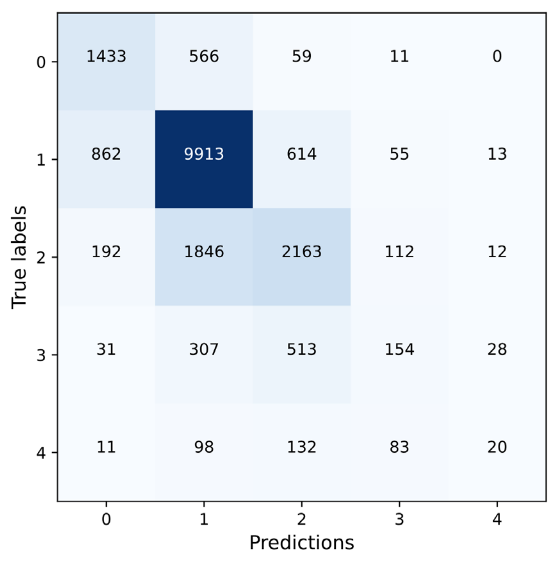
where P_i, R_i is the precision and recall on the *i*-th label respectively, and w_i is the proportion of true instances for the *i*-th label. The results on *$OP* slot are collected from the generated **WHERE** conditions that hit the correct properties. According to Fig.4, the labels of these subtasks are extremely imbalanced. As seen, the great majority of the HQL queries do not have a property or an aggregator in their **SELECT** clauses, or have only one condition in their **WHERE** clauses. Besides, the "=" operator is used far more frequently than other operators. Despite of the imbalanced labels, HQLgen shows good classification performance according to Table 3, where its weighted macro-F1 scores are all higher than 65%. Fig.4 (a) shows that HQLgen is able to identify 59% (2,306) of the queries that actually choose a property via their **SELECT** clauses. The most used aggregate function is COUNT, followed by MAX and SUM, where HQLgen can correctly predict 55.4% (643), 33.3% (83) and 33.8% (44) of them respectively. On contrary, AVG and MIN are seldomly applied so the model can hardly know when to use them. According to Fig.4 (c), the model tends to miss conditions when actual *#COND* is larger than 2, indicating that the model has difficulty on generating complex **WHERE** clauses. ">", "<", "IN" and "LIKE" are the most used operators other than "=". From Fig.4 (d), we can find that our model is able to tell ">" and "<" apart from other operators but has some trouble in distinguishing these two. On the other hand, "!=", "IS NULL", "IN" and "LIKE" are always confused with "=", showing the challenges in predicting them. We will discuss why HQLgen fails via analyzing its output examples in Section 7. Overall, Fig.4 and Table 3 suggests out approach could be improved by better handling the imbalanced labels of *#SEL*, *$AGGR*, *#COND* and *$OP*, e.g., by using sampling techniques.



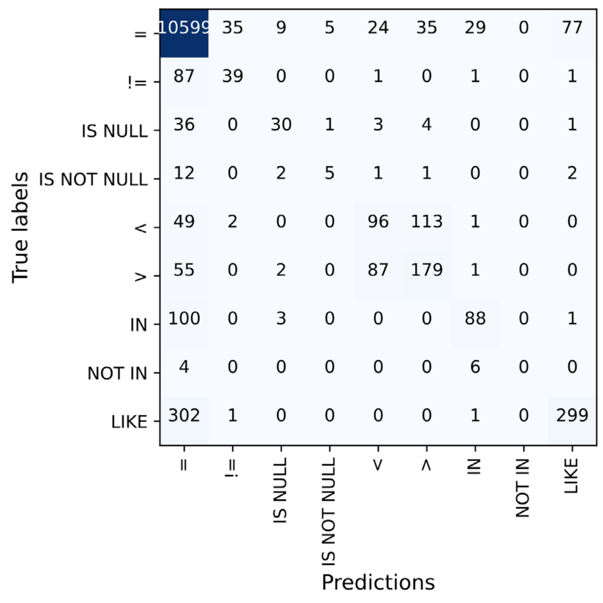
(a) Predictions on *#SEL*



(b) Predictions on *$AGGR*



(c) Predictions on *#COND*



(d) Predictions on *$OP*

Fig. 4 The confusion matrices of the predictions (with mixed data splitting) on *#SEL*, *$AGGR*, *#COND* and *$OP*.

Table 3 The weighted macro F1 scores on *#SEL*, *$AGGR*, *#COND* and *$OP*

Splitting	*#SEL*	*$AGGR*	*#COND*	*$OP*
Mixed	85.62%	94.20%	69.33%	90.43%
Cross-Project	80.29%	92.35%	65.43%	86.92%

Table 4 The impact of method comment on performance

Splitting	Comment	ACC	ACC$_{\text{SEL}}$	ACC$_{\text{PROP\_S}}$	ACC$_{\text{AGGR}}$	ACC$_{\text{WHERE}}$	ACC$_{\text{\#COND}}$	M$_{\text{COND}}$
Mixed	True	**34.73%**	**81.98%**	**84.52%**	**94.92%**	**40.12%**	**70.13%**	**52.51%**
	False	29.83%	78.03%	80.42%	94.09%	35.40%	67.29%	51.32%
Cross-Project	True	25.43%	**74.42%**	76.47%	92.99%	30.42%	63.74%	43.73%
	False	**26.16%**	74.24%	**76.96%**	**93.85%**	**31.67%**	**64.05%**	**45.81%**

7 Discussion

7.1 Effectiveness of Comments

Since code comments are considered crucial in program comprehension (Sridhara et al. 2010), we discuss the impact of comments on the performance of HQLgen in this section. Note that only 17% (3,268) of our target methods are commented, so compute the metrics on the whole dataset is not reasonable. Therefore, we train HQLgen with and without method comments via the settings described in Section 6.1, and then collect the predictions on these examples. The results are shown in Table 4. With mixed data splitting, taking comments as inputs significantly improves the scores on all metrics. In particular, ACC is increased by 4.9% compared to using method signatures only. Since most comments describe the functionality of the target method, they are helpful when generating the executed queries. However, using no comments performs slightly better with cross-project splitting. Actually, only a few projects contain commented target methods, so the model will fail to learn how to leverage the comments if these projects are assigned to the test set. For future work, it is worth trying to use the comments in the call context and target class.

7.2 Qualitative Analysis

We illustrate four good predictions plus four flawed ones generated by HQLgen in Table 5 and Table 6 respectively for qualitative analysis. For each of these examples, the target method, call context and number of candidate properties (#Properties) are shown together. The ground truth queries in the target method are in italic and marked in red. Due to limited space, we show part of target method body for Example 1, 5, 7, and part of call context for Example 4.

In Example 1, the method parameters are passed to the HQL query and they have the same names and types as the corresponding properties. Meanwhile, they also appear in the call context of the target method. In this case, the proposed model successfully captures this information and generates the correct query. Example 2 is more challenging in that the parameter name "user_id" does not occur in the candidate properties and the call context is not helpful. Moreover, the desired property "id" is nested in the object "gebruikerid" but not defined directly by the target class (actually, there is a property "Advertentie.id" that may confuse the model). Thanks to our input representations, the token "Gebruiker" in the method name is separately embedded in the input sequence so that the property "gebruikerid" is highlighted when computing the attention weights via (3), and HQLgen eventually produce the matched query. By contrast, the rule-based baseline fails in this case. Example 3 further illustrates the effectiveness of the representations of parameter and property. Using camelCase and "." to split the inputs, the method parameters "categoryId"

Table 5 Good predictions from HQLgen

Example 1	

Target Method	public User verifyUser(String username, String password) {
	    Session session = sessionFactory.getCurrentSession();
	    Query query = session.createQuery(*"FROM User as u where u.username = :username and u.password = :password"*).setString("username", username).setString("password", password);
	    User user = (User) query.uniqueResult();
	    ……
	}

Called In	public ExtResult loginUser(String username, String password)
#Properties	21
Prediction	SELECT User FROM User WHERE User.username = :value AND User.password = :value
Example 2	

Target Method	@Override
	public List<Advertentie> getAdvertentieFromGebruiker(Integer user_id) {
	    return em.createQuery(*"SELECT a FROM Advertentie a WHERE a.gebruikerid.id ='" + user_id + "'"*).getResultList();
	}

Called In	protected void processRequest(HttpServletRequest request, HttpServletResponse re-sponse)
#Properties	23
Prediction	SELECT Advertentie FROM Advertentie WHERE Advertentie.gebruikerid.id = :value
Example 3	

Target Method	@SuppressWarnings("unchecked")
	@Override
	@Transactional
	public List<Product> findByCategoryAndProducer(int categoryId, int producerId) {
	    return sessionFactory.getCurrentSession().createQuery(*"from Product p where p.producer.id = :producerId and p.category.id = :categoryId"*).setParameter("pro-ducerId", producerId).setParameter("categoryId", categoryId).list();
	}

Called In	public List<Product> findByCategoryAndProducer(int productCategoryId, int producerId)
	public String getProductsByCategory(Integer categoryId, Map<String, Object> map)
#Properties	58
Prediction	SELECT Product FROM Product WHERE Product.category.id = :value AND Product.producer.id = :value
Example 4	

Target Method	@Override
	public Long getCount(Marka marka) {
	    Query query = getSession().createQuery(*"SELECT COUNT(*) FROM Termek t WHERE t.marka = :marka"*);
	    query.setParameter("marka", marka);
	    return (Long) query.list().get(0);
	}

Called In	public Long getTermekCount(Tipus tipus, Marka marka)
	public Long getRaktarCount(Tipus tipus)
	public Long getRaktarCount(Tipus tipus, Marka marka)
	public Long getCount(Integer statusz)
	public ActionForward execute(ActionMapping mapping, ActionForm form, HttpS-ervletRequest request, HttpServletResponse response)
#Properties	21
Prediction	SELECT COUNT (Termek) FROM Termek WHERE Termek.marka = :value

Table 6 Incorrect predictions from HQLgen

Example 5	

Target Method	`@Override`
	`public BigDecimal getTotalByProAndSupp(Long proId, Long suppId) {`
	`    String hql = "select sum(amount) from Bill where projectNew.id=? and suppli-ersAssess.suppliersId=?";`
	`    Query query = getSession().createQuery(hql);`
	`    query.setParameter(0, proId);`
	`    query.setParameter(1, suppId);`
	`    ……`
	`}`

Called In	`public Map<String, Object> getApplyDataByProNo(String proNo, String materialId, Long proId, Long suppId)`
	`public BigDecimal getTotalByProAndSupp(Long proId, Long suppId)`
	`public String getPayedByProNo()`
#Properties	126
Prediction	`SELECT SUM (Bill.amount) FROM Bill WHERE Bill.processRunId = :value AND Bill.project-New.id = :value`
Example 6	

Target Method	`public List<Video> findVideosByName(String videoName) {`
	`    TypedQuery<Video> query = entityManager.createQuery("SELECT v from Video v where v.videoName LIKE :videoName", Video.class);`
	`    query.setParameter("videoName", "%" + videoName + "%");`
	`    return query.getResultList();`
	`}`

Called In	`public List<Video> getNameFilteredVideos()`
#Properties	19
Prediction	`SELECT Video FROM Video WHERE Video.videoName = :value`
Example 7	

Target Method	`private void buildWordCount(@NonNull StringBuilder result, @NonNull Player p, @Nul-lable Round roundItem) {`
	`    if (roundItem == null) {……}`
	`    else {`
	`        Object count = em.createQuery("SELECT count(w) FROM Word w WHERE w.player = ?1 AND w.round = ?2 AND w.disabled = ?3").setParameter(1, p).setParameter(2, roundItem).setParameter(3, new Boolean(false)).getSingleResult();`
	`    ……}`
	`}`

Called In	`private void buildPlayer(StringBuilder result, Player p, Room roomItem, Round roundItem, boolean roundC, boolean isCurrentPlayer)`
#Properties	20
Prediction	`SELECT COUNT (Word) FROM Word WHERE Word.player = :value`
Example 8	

Target Method	`public List getTimeSlotEntries(String resourceID, boolean isParticipant, long from, long to) {`
	`    if (to <= 0)`
	`        to = Long.MAX_VALUE;`
	` return _persister.createQuery("FROM CalendarEntry AS ce " + "WHERE ce.resourceID IN (:idlist) " + "AND ce.startTime < :end AND ce.endTime > :start " + "ORDER BY ce.startTime").setParameterList("idlist", createIDListForQuery(resourceID, isPartic-ipant)).setLong("start", from).setLong("end", to).list();`
	`}`

Called In	`public List getTimeSlotEntries(AbstractResource resource, long from, long to)`
#Properties	9
Prediction	`SELECT CalendarEntry FROM CalendarEntry WHERE CalendarEntry.resourceID = :value AND CalendarEntry.startTime > :value`

and “producerId” have exactly the same embeddings as the properties “category.id” and “producer.id”, so they are successfully selected by HQLgen. Note that these parameters also appear in the call context. The call context is even more useful in Example 4, since the frequently appeared token “count” and “marka” in it are important clues for the model to choose the corresponding aggregator and property. Therefore, taking call context into account is reasonable.

Table 6 illustrates the other four cases that HQLgen typically fails to generate matched results. In Example 5, the target method parameters are implicitly named via abbreviations, and the target class contains up to 126 candidate properties. We can find that HQLgen mistakenly links the parameter “proId” to “Bill.projectNew.id” and misses the correct property “Bill.suppliersAssess.suppliersId” associated with “suppID”. To better handle such identifiers, a possible solution is to complete the abbreviations according to their context before feeding them to the model. Fortunately, HQLgen is still able to predict the correct aggregator SUM. This denotes that it successfully captures the semantics of “getTotal” in the method names. Example 6 shows the challenge on identifying the operators. The generated query is almost correct except that the model confuses the operator “LIKE” with “=”. Unfortunately, even human participant can hardly determine “LIKE” is going to be used according to the provided context, especially for the poorly named parameter “videoName”. To achieve pattern-matching via LIKE, a “%” must be included in the “videoName” parameter, and thus this parameter is a pattern expression rather than the name of a video. Similarly, the method name “findVideosByName” is also inappropriate since it indicates this method is to accurately find a video by its name instead of acting like a search engine. Moreover, this query is affected a common SQL anti-pattern called “poor man’s search engine” since such queries cannot always benefit from indexes (Nagy and Cleve 2017), which should be avoided in practice.

Sometimes *#COND* is also challenging to predict, as shown in Example 7. There are two difficulties in predicting correct *#COND* here: First, the method parameter “result” is not for the HQL query; Second, the property “disable” in the ground truth is not mentioned in the context. Although HQLgen performs well on choosing the aggregator “COUNT” and the property “player”, it misses the other two **WHERE** conditions. The HQL query in Example 8 is more complicated, where the model is able to choose the correct properties “resourceID” and “startTime” but fails to assign them correct operators. Checking the target method body, the actual value passed to the placeholder “idlist” is a result of API call that takes “resourceID” and “isParticipant” as input. With the given context, it is impossible for the model to find out that the value of “resourceID” is a list, and thus it is no way to correctly yield the operator “IN”. On the other hand, the given context does not provide enough details to predict the operators for “startTime” and “endTime” as well. From our understanding, this project is affected by architecture erosion (Li et al. 2021) (i.e., boundaries disappear among layers and components) since it involves business logic of generating candidate values for query in the persistence code. Such actions should be performed in the call context of the target method, and the input of the target method should be prepared accordingly. Example 7 and 8 suggest that more project-related knowledge is necessary for the generation of complex queries, e.g., the detection of architecture erosion could be helpful for capturing potential business logic in the data persistence code. In summary, according to Table 6, although the generated queries are not perfect in all scenarios, they could be helpful to the developers since they will work as expected after simple modifications.

Table 7 Performance on different embedding size d_e and RNN hidden size d_h

d_e (d_h = 32)	ACC	d_h (d_e = 96)	ACC
16	31.23%	16	31.45%
32	32.54%	32	**34.52%**
64	33.64%	64	32.92%
96	**34.52%**	96	31.52%
128	33.86%	128	31.75%

7.3 Hyperparameter Selection & Time Cost

The most important hyperparameter of HQLgen is the dimension of name/type embedding d_e and the hidden size of the LSTM encoder d_h. To check their impact on model performance, we tune them from 16 to 128 and evaluate HQLgen with mixed data splitting described in Section 6. When adjusting either d_e or d_h, other hyperparameters remain unchanged. The obtained ACC scores are shown in Table 7. As seen, increasing d_e continuously improves ACC until it reaches 96, while the best d_h is 32. This indicates that HQLgen prefers relatively large embedding size but small encoder size. Intuitively, compressing the embedded vectors into lower dimensions forces the model to extract important information from inputs, and the reduced vectors could make the downstream classifications easier. Another reason why smaller d_h works better could be that the input sequences of HQLgen are short so large RNN encoders are not required. The size of both **SELECT** and **WHERE** predictors is 64MB, resulting in a total model size of 128MB. One training epoch of HQLgen takes around 46 seconds, while predicting a batch of testing examples (the batch size is 32) only needs 0.09 second. Overall, the proposed approach is applicable.

7.4 Threats to Validity

The first threat to validity our approach to locate projects containing HQL queries. We use the criteria proposed by Nagy et al. (2015b) to locate projects containing HQL queries, i.e., locating the createQuery method calls. However, there exists other implementations to write and define HQL queries, such as writing queries in the @Query annotation provided by Spring Data JPA. We evaluated this implementation using BOA, and we find only 621 projects were using this approach, which account for less than 3% of all available projects, and thus we do not take them into account.

The second threat to validity is the dataset we built. Although we use certain rules to filter the extracted projects and HQL queries, there could still be noisy examples such as poorly named identifiers (e.g. “arg”, “Main”) and the ones within test projects. Moreover, the quality of the project in our dataset varies. We are not able to discriminate the quality of projects by simply looking at the GitHub stars, because most of them do not have stars since they are designed for organizational use or specific purposes, which is less attractive to public than popular infrastructural projects or libraries developed by major open-source foundations (e.g., ASF, Eclipse). In future work, we may use static code quality analyzer and exploit manual inspection to generate a finer-grained dataset.

The third threat to validity is that we do not consider execution accuracy. Since the quality of some projects are not guaranteed, we cannot compile every project and generate their related database structure in the dataset. However, we believe we could include such analysis in the up-mentioned finer-grained dataset.

The fourth threat is that we did not involve human study. We consult 2 professional Java Web developers with more than 5 years' experience on Hibernate ORM development about the quality of our generated queries. However, we find that they are very unconfident with their suggestion since they are not familiar with the business logic, context, and data schema of the software projects. Thus, we believe that using the original query as the gold standard with ACC as the indicator is reasonable.

8 Conclusion

In this paper, we introduce the task of automatically generating HQL queries from program context in order to assist the developers when writing data persistence codes. To address this problem, we propose a novel model named HQLgen based on deep learning and template filling. It embeds the context of a desired HQL query into vectors and encodes them via RNN, and then predicts the key elements in the query via attention mechanism and MLP. To construct the dataset for training and evaluation, we locate and extract projects containing HQL queries in GitHub followed by extensive cleaning and preprocessing. Experimental results show that the proposed approach achieves an accuracy of 34.52% on generating simple HQL queries, and its outputs could be helpful to the developers. The collected dataset is also made publicly available. Despite that we limit the scope on the generation of simple queries, the results obtained by HQLgen suggest there is still much room for improvement on this task. For future work, we argue that it is necessary to identify, extract and leverage more useful information from the program context rather than turn to complex HQL queries. We will also try to incorporate useful rules into HQLgen to make it more robust. Moreover, we will extend our context to ORMs in other programming languages such as SQLAlchemy in Python.

References

Androutsopoulos, I., Ritchie, G., Thanisch, P.: Masque/sql: An efficient and portable natural language query interface for relational databases. In: Proceedings of the 6th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems. IEA/AIE'93, pp. 327–330. Gordon amp; Breach Science Publishers (1993)

Androutsopoulos, I., Ritchie, G.D., Thanisch, P.: Natural language interfaces to databases - an introduction. Nat. Lang. Eng. 1(1), 29–81 (1995). https://doi.org/10.1017/S135132490000005X

Bogin, B., Gardner, M., Berant, J.: Global reasoning over database structures for text-to-sql parsing. In: Inui, K., Jiang, J., Ng, V., Wan, X. (eds.) Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019, pp. 3657–3662. Association for Computational Linguistics (2019). https://doi.org/10.18653/v1/D19-1378

Chen, T.-H., Shang, W., Hassan, A.E., Nasser, M.N., Flora, P.: Cacheoptimizer: helping developers configure caching frameworks for hibernate-based database-centric web applications. In: Proc. 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pp. 666–677 (2016)

Chen, T.-H., Shang, W., Yang, J., Hassan, A.E., Godfrey, M.W., Nasser,M., Flora, P.: An empirical study on the practice of maintaining object-relational mapping code in Java systems. In: Proc. 13th International Conference on Mining Software Repositories (MSR), pp. 165–176 (2016)

Cho, K., van Merrienboer, B., Gu¨l¸cehre, C¸., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: Moschitti, A., Pang, B., Daelemans, W. (eds.) Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A Meeting of SIGDAT, a Special Interest Group of the ACL, pp. 1724–1734. ACL (2014). https://doi.org/10.3115/v1/d14-1179

Cook, W.R., Greene, R., Linskey, P., Meijer, E., Rugg, K., Russell, C., Walker, B., Wittig, C.: Objects and databases: State of the union in 2006. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pp. 926–928 (2006)

Di Giacomo, M.: MySQL: Lessons learned on a digital library. IEEE Software 22(3), 10–13 (2005)

Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: Ultra-large-scale software repository and source-code mining. ACM Trans. Softw. Eng. Methodol. 25(1), 7:1–7:34 (2015)

Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Teh, Y.W., Titterington, D.M. (eds.) Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010. JMLR Proceedings, vol. 9, pp. 249–256. JMLR.org (2010)

Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. 9(8), 1735–1780 (1997). https://doi.org/10.1162/neco.1997.9.8.1735

Iyer, S., Konstas, I., Cheung, A., Krishnamurthy, J., Zettlemoyer, L.: Learning a neural semantic parser from user feedback. In: Barzilay, R., Kan, M. (eds.) Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers, pp. 963–973. Association for Computational Linguistics (2017). https://doi.org/10.18653/v1/P17-1089

Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015)

Li, F., Jagadish, H.V.: Constructing an interactive natural language interface for relational databases. Proc. VLDB Endow. 8(1), 73–84 (2014). https://doi.org/10.14778/2735461.2735468

Li, R., Liang, P., Soliman, M., Avgeriou, P.: Understanding architecture erosion: The practitioners' perceptive. In: Proc. IEEE/ACM 29th International Conference on Program Comprehension (ICPC), pp. 311–322 (2021)

Loli, S., Teixeira, L., Cartaxo, B.: A catalog of object-relational mapping code smells for Java. In: Proc. 34th Brazilian Symposium on Software Engineering (SBES), pp. 82–91 (2020)

Meurice, L., Nagy, C., Cleve, A.: Detecting and preventing program inconsistencies under database schema evolution. In: Proc. IEEE 16th International Conference on Software Quality, Reliability and Security (QRS), pp. 262–273 (2016)

Nagy, C., Cleve, A.: Mining stack overflow for discovering error patterns in SQL queries. In: Proc. IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 516–520 (2015)

Nagy, C., Cleve, A.: A static code smell detector for SQL queries embedded in Java code. In: Proc. IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 147–152 (2017)

Nagy, C., Meurice, L., Cleve, A.: Where was this SQL query executed? A static concept location approach. In: Proc. IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 580–584 (2015)

Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: Fu¨rnkranz, J., Joachims, T. (eds.) Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel, pp. 807–814. Omnipress (2010)

Nazario, M.F.C., Guerra, E., Bonifacio, R., Pinto, G.: Detecting and reporting object-relational mapping problems: An industrial report. In: Proc. ACM/IEEE 13th International Symposium on Empirical Software

Engineering and Measurement (ESEM), pp. 1–6 (2019)

Popescu, A., Etzioni, O., Kautz, H.A.: Towards a theory of natural language interfaces to databases. In: Leake, D.B., Johnson, W.L., Andr´e, E. (eds.) Proceedings of the 8th International Conference on Intelligent User Interfaces, IUI 2003, Miami, FL, USA, January 12-15, 2003, pp. 149–157. ACM (2003). https://doi.org/10.1145/604045.604070

Presler-Marshall, K., Heckman, S., Stolee, K.: SQLRepair: Identifying and repairing mistakes in student-authored SQL queries. In: Proc. IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), pp. 199–210 (2021)

Procaccianti, G., Lago, P., Diesveld, W.: Energy efficiency of ORM approaches: An empirical evaluation. In: Proc. 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 36–13610 (2016)

Silva, T.M., Serey, D., de Figueiredo, J.C.A., Brunet, J.: Automated design tests to check hibernate design recommendations. In: Proc. 33rd Brazilian Symposium on Software Engineering (SBES), pp. 94–103 (2019)

Singh, R., Bezemer, C., Shang, W., Hassan, A.E.: Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm. In: Proc. 7th ACM/SPEC International Conference on Performance Engineering (ICPE), pp. 309–320 (2016)

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L.L., Vijay-Shanker, K.: Towards automatically generating summary comments for java methods. In: Pecheur, C., Andrews, J., Nitto, E.D. (eds.) ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010, pp. 43–52. ACM (2010). https://doi.org/10.1145/1858996.1859006

Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learn- ing with neural networks. In: Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N.D., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada, pp. 3104–3112 (2014)

Vial, G.: Lessons in persisting object data using object-relational mapping. IEEE Software 36(6), 43–52 (2019)

Wang, B., Shin, R., Liu, X., Polozov, O., Richardson, M.: RAT-SQL: relation-aware schema encoding and linking for text-to-sql parsers. In: Jurafsky, D., Chai, J., Schluter, N., Tetreault, J.R. (eds.) Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020, pp. 7567–7578. Association for Computational Linguistics (2020). https://doi.org/10.18653/v1/ 2020.acl-main.677

Warren, D.H.D., Pereira, F.C.N.: An efficient easily adaptable system for interpreting natural language queries. Am. J. Comput. Linguistics 8(3-4), 110–122 (1982)

Xu, X., Liu, C., Song, D.: Sqlnet: Generating structured queries from natural language without reinforcement learning. CoRR abs/1711.04436 (2017)

Yaghmazadeh, N., Wang, Y., Dillig, I., Dillig, T.: Sqlizer: query synthesisfrom natural language. Proc. ACM Program. Lang.1(OOPSLA), 63–16326 (2017). https://doi.org/10.1145/3133887

Yu, T., Li, Z., Zhang, Z., Zhang, R., Radev, D.R.: Typesql: Knowledge-based type-aware neural text-to-sql generation. In: Walker, M.A., Ji, H., Stent, A. (eds.) Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers), pp. 588–594. Association for Computational Linguistics (2018). https://doi.org/10.18653/v1/n18-2093

Yu, T., Yasunaga, M., Yang, K., Zhang, R., Wang, D., Li, Z., Radev, D.R.: Syntaxsqlnet: Syntax tree networks for complex and cross-domain text-to-sql task. In: Riloff, E., Chiang, D., Hockenmaier, J., Tsujii, J. (eds.) Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018, pp. 1653–1663. Association for Computational Linguistics (2018). https://doi.org/10.18653/v1/d18-1193

Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., Radev, D.R.: Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In: Riloff, E., Chiang, D., Hockenmaier, J., Tsujii, J. (eds.) Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018, pp. 3911–3921. Association for Computational Linguistics (2018). https://doi.org/10.18653/v1/d18-1425

Zhong, V., Xiong, C., Socher, R.: Seq2sql: Generating structured queries from natural language using reinforcement learning. CoRR abs/1709.00103 (2017)