International Journal of Advanced Engineering Research and Science (IJAERS) **[Vol-6, Issue-4, Apr- 2019]**
https://dx.doi.org/10.22161/ijaers.6.4.35 **ISSN: 2349-6495(P) | 2456-1908(O)**

Detecting anti-patterns in SQL Queries using Text Classification Techniques

Abdou Rahmane Ousmane, Hongwei Xie*

College of Computer Science and Technology, Taiyuan University of Technology, Taiyuan, Shanxi, China (030024)

Abstract— *A major problem with using relational databases, is writing efficient SQL queries. Some common errors known as anti-patterns are frequent in SQL queries and can seriously impact the query execution time and sometimes, the database general performance. This paper shows how ma-chine learning techniques can be lever-aged to detect anti-patterns in SQL queries by approaching the problem as a text classification problem. Our result is a model based on a convolutional neural net-work that can be used to classify a SQL query into zero, one or many anti-patterns classes.*

Keywords— SQL, relational database, text classification techniques.[1]

I. INTRODUCTION

With the increasing amount of information stored in relational databases, it is necessary to write SQL queries that execute faster. Anti-patterns in SQL are common mistakes that if avoided, can make a query executes faster. For example, when query-ing an indexed column, replacing the OR operator with the IN operator, will result in better perfor-mance, because the IN operator leverages the in-dex. Thus, using the OR operator in this case, is an anti-pattern.

```
SELECT   url FROM pictures WHERE
    id = 10 OR id    = 20
```

can be rewritten as

```
SELECT   url FROM pictures WHERE
    id IN  (10,20)
```

By detecting the anti-patterns in a query, we can rewrite it into a better version. In this paper, we approach the problem as a multi-class multi-label classification problem. Our solution is schema-independent, meaning that the decision made by the neural network doesn't depend on the database logical or physical structure. The dataset used has been built from SQL queries provided by Sky-Server from Sloan Digital Sky Survey (SDSS).

SkyServer, the portal from the SDSS catalog, provides data access tools for astronomers and sci-entific education. Through SkyServer, users can use the SQL language to query the Sloan Digi-tal Server database. Since 2001, the portal has seen more than 280 million SQL queries submit-ted by users and those queries have been opened to the public through the different data releases. We fetch 1 million queries from SkyServer, that we filter, process and transform. The final dataset of usage contains 363616 unique SELECT queries.

Following a supervised learning approach, the SQL queries from SkyServer are used as input data; we manually label the data by associating each SQL query with a list of anti-patterns it con-tains.

Our model is based on a convolutional neural network trained to classify a query into multiple categories. We use the one-hot encoding technique to encode the queries as word vectors. For encoding the anti-patterns classes we use a one dimensional tensor with each class represented as an in-teger.

We explore some of the important work in the field of SQL anti-patterns detection in section 2. In section 3, we explain in details the process fol-lowed to build the dataset. Then, we discuss our model architecture in section 4. In section 5, we analyze the results from our experiments. Finally in the conclusion, we compare our work to the existing solutions and explore the possible future work.

II. RELATED WORK

Common mistakes in SQL has been already in the interest of researchers before the appearance of the ISO SQL-92 standard. In 1985, Welty studied how human factors can affect users in using SQL and found that user performance could be significantly improved. Later, Brass et al. started working on the automatic detection of logical errors in SQL queries and extended their work with the recognition of common semantic mistakes. They implemented the

[1] *This work is supported by the Dynamic Analysis and Synchronous Research of Complex Biological Networks project from the National Natural Science Foundation in China (No: 61702356, 2018-01-01 - 2020-12-31) and by the ToF depth camera image real-time depth denoising re-search project from Shanxi Science and Technology Depart-ment(No: 201801D121143, 2019-01-01 - 2020-12-30)*

*Corresponding author: xiehongwei@tyut.edu.cn

SQLLint tool which was able to au-tomatically identify these errors in (syntactically correct) SQL statements. The tool seems to be unsupported today. There is another online tool named SQLLint, but it is a SQL beautifier.

There are also books in this area. The Art of SQL and Refactoring SQL Applications pro-vide guidelines to write efficient queries, while the book of Bill Karwin collects antipatterns that should be avoided.

In a paper, Ahadi et al.,presented a large-scale analysis of students semantic mistakes in writing SQL SELECT statements. They collected data from over 2,300 students across nine years and summarized typical mistakes of the students. They found that most of the mistakes were made in queries which require a JOIN, a subquery or a GROUP BY operator. We argue that queries typ-ically use more complex syntax in practice com-pared to student projects. Hence, the situation can be even worse.

In the realm of embedded SQL, Christensen et al. proposed a technique and a tool (JSA, Java String Analyzer) to extract string expressions from Java code statically. As a potential application of their approach, they check the syntax of dynami-cally generated SQL strings. They limit their ap-proach to the syntactic validation of the queries.

Wassermann et al. propose a static string analy-sis technique to identify possible errors in dynam-ically generated SQL code. With the implemen-tation of a CFL-reachability algorithm they detect type errors (e.g., concatenating a character to an integer value). Their approach works with ex-tracted query strings of valid SQL syntax. In a tool demo paper, they present their prototype tool called JDBC Checker.

Recently, Anderson and Hills studied query construction patterns in PHP. They analyzed query strings embedded in PHP code with the help of the PHP AiR framework.

Quality assessment of embedded SQL was pro-posed by Brink et al. in 2007. They analyzed em-bedded query strings in PL/SQL, Cobol, and Vi-sual Basic programs while they propose a generic approach which could be applied to Java too. They investigate relationships which could be detected through embedded queries (e.g., access, dupli-cation, control dependencies) and they propose quantitative query measures for quality assess-ment.

Many static techniques which try to deal with embedded query strings do it with the purpose of SQL injection detection. Yeole and Meshram pub-lished a survey of these techniques. SQL injection detection is different as the goal is specifically to determine whether a query could be affected by user input.

Some papers also tackle SQL fault localization techniques. A dynamic approach was proposed by Clark et al. to localize SQL faults in database applications. They provide command-SQL tuples to show the SQL statements executed at database-interaction points.

A recent work of Delplanque et al. targets the database to assess the quality of the schema and to detect design smells in it. They implement a tool called DBCritics which can analyze PostgreSQL schema dumps and identify design smells such as missing primary keys or foreign key references.

A tool which also has to be mentioned here is the Eclipse plugin called Alvor and JSA [17], this plug-in analyzes the string expressions in Java code. What is more, Alvor checks syntax correct-ness, semantics correctness, and object availability by comparing the extracted queries against its in-ternal SQL grammar and by checking SQL state-ments against an actual database.

III. DATASET

3.1 Collecting the queries

We start building our dataset, by fetching 1 mil-lion successful SQL queries from the SkyServer catalog.

```
SELECT TOP 1000000      statement
FROM SqlLog
WHERE error  = 0
```

Some of these queries need to be filtered out, in order to build a more focused dataset.

3.2 Filtering

From the fetched queries, we remove the dupli-cates, so the dataset contains unique queries only.

```
allQueries = list(set(values))
```

As we focus on query anti-patterns, we remove all of the non SELECT queries.

```
import re
allQueries = list(
    filter(
        lambdaitem: re.search(
            ”ˆselect”,
            item.lower()
        ),
        allQueries
    )
)
```

In the end, the dataset is reduce from 1000000 to 318188 queries.

3.3 Transforming

In order to eliminate irrelevant information and reduce the size of our dataset vocabulary, we re-place all of the schema-related terms contained in the queries with standard words. Thus the queries contain almost only standard SQL keywords.

```
SELECT name from students;
    will be transformed to
SELECT column from table;
```

3.4 Annotating

Following a supervised learning approach, having SQL queries as input, we need to map each query to a set of anti-patterns as labels.

Our work is based on 16 common anti-patterns. To each of the query, we match a single anti-pattern. In fact, a single query can contain several anti-patterns, but for simplicity purpose, we only consider the most dominant anti-pattern. We ex-plain in detail each anti-pattern in the Appendix section.

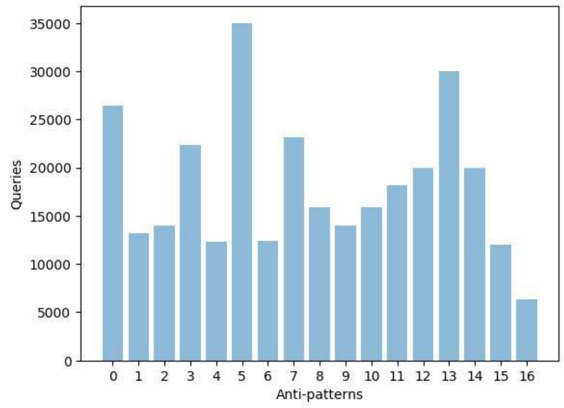

Fig.1: Dataset visualization

IV. MODEL

4.1 Data Encoding

4.1.1 Queries

Each SQL query in our current dataset is a list of words. Word representation methods gener-ally fall into two categories. The first consists of methods such as one-hot vectors. This method is problematic due to homonymy and polysemy words. The other category consists of using un-supervised learning method to obtain continuous word vector representations. Recent research re-sults have demonstrated that continuous word rep-resentations are more powerful.

In this paper, we use word embedding based on word2vec (Mikolov et al., 2013). To encode the SQL queries of our dataset, we choose to use the pre-trained google word2vec embedding. The model is trained on 100 billion words from Google News by using the Skip-gram method and maxi-mizing the average log probability of all the words using a softmax function. Our result model con-tains 123.852 tokens.

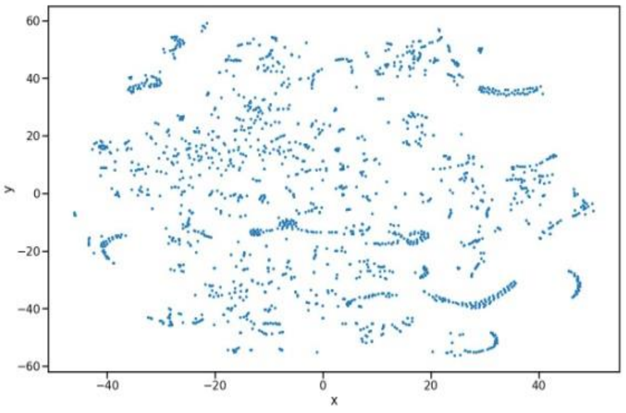

Fig.2: Queries Embedding

0	SELECT *
1	NULL Usage
2	NOT NULL Usage
3	String Concatenation
4	GROUP BY Usage
5	ORDER BY RAND Usage
6	Pattern Matching Usage
7	Spaghetti Query Alert
8	Reduce Number of JOINs
9	Eliminate Unnecessary DISTINCT
10	Implicit Column Usage
11	HAVING Clause Usage
12	Nested sub queries
13	OR Usage
14	UNION Usage
15	DISTINCT & JOIN Usage
16	No anti-pattern

Fig.3: Anti-patterns list

4.1.2 Anti-patterns

As our work is based on 16 Fixed anti-patterns, we encode the label data as 1D Vector of integers.

4.2 Convolutionnal Neural Network

The convolution neural network is a state-of-the-art method to model semantic representations of sentences. The convolution action has been com-monly used to synthesize lexical n-gram informa-tion. In our model, we use three different convo-lutional filters with varying convolution window size to form parallel CNNs so that they can learn multiple type of embedding of local regions so as to complement each other to improve model accu-racy. The final output is the concatenation of the output of each.

V. EXPERIMENTS

5.1 Settings

For all our experiments, we use the Stochastic Gradient Descent optimization algorithm with a learning rate of 0.1 and a weight decay of 0.95. We conduct the experiments with 50 epochs and we use mini-batches of size 64. We evaluate the model every 100 steps. We use google pre-trained word2vec thus the dimension of each word vector is 300.

We study the sensivity of the proposed model to the convolutional region size, the number of con-volutional feature and the dropout rate. We found that we achieve the best performance when we use the settings values listed in the Table I.

Our model is developed in Python with Tensor-flow and Numpy libraries. The experiments are conducted on a MAC OS PC with 2.9 GHz Intel Core i7 processor and 8 GB RAM.

Region size	Feature Maps	Dropout rate
(4, 5, 6)	150	0.4

Fig.4: Experimental Settings

5.2 Validation method

For validating our model we use the iterated K-Fold validation model.

The dataset is split into 10 mini-datasets, which are used to validate each subset repeatedly.

5.3 Results

We compare our results with Sqlcheck . Sqlcheck is a lint tool that relies on syntax checking logic, to detect anti-patterns in SQL queries. We run SQL check on each of our dataset query, and store the results, which we then compare to our CNN re-sults.

SqlCheck	Our model
80	83.2

Fig.5: Experimental Settings

After running the experiments, our model can detect anti-pattern in a query with an accuracy of 83.2.

VI. CONCLUSION AND FUTURE WORK

In this work, we experimented using text classi-fication techniques to detect anti-patterns in SQL queries. The model uses a neural network with a custom dataset built from SkyServer catalog SQL queries. Experimental results demonstrate that, our model is quite accurate and can outperform lint syntax checking software.

For the future, we could focus on rewriting queries based on the anti-patterns detected.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my super-visor and corresponding author on this work, Xie Hongwei PhD, who has been a great mentor dur-ing the years of research.

I would also like to thank my friend Miguel Kakanakou, for his very helpful advice and ex-perience sharing during the whole process of this work.

APPENDIX

Anti-patterns explanation select *

When you SELECT *, you're often retrieving more columns from the database than your appli-cation really needs to function. This causes more data to move from the database server to the client, slowing access and increasing load on your ma-chines, as well as taking more time to travel across the network.

Consider a scenario where you want to tune a query to a high level of performance. If you were to use *, and it returned more columns than you actually needed, the server would often have to perform more expensive methods to retrieve your data than it otherwise might.

When you SELECT *, it's possible to retrieve two columns of the same name from two different tables. This can often crash your data consumer null usage

NULL is not the same as zero. A number ten greater than an unknown is still an unknown. NULL is not the same as a string of zero length. Combining any string with NULL in standard SQL returns NULL. NULL is not the same as false. Boolean expressions with AND, OR, and NOT also produce results that some people find confusing not null usage

When we declare a column as NOT NULL, it should be because it would make no sense for the row to exist without a value in that column.

string concatenation

You may need to force a column or expression to be non-null for the sake of simplifying the query logic, but you don't want that value to be stored. Use COALESCE function to construct the con-catenated expression so that a null-valued column doesn't make the whole expression become null.

group by usage

Every column in the select-list of a query must have a single value row per row group.

order by rand usage

Sorting by a nondeterministic expression (RAND()) means the sorting cannot benefit from an index

pattern matching usage

The most important disadvantage of pattern-matching operators is that they have poor per-formance. A second problem of simple pattern-matching using LIKE or regular expressions is that it can find unintended matches.

spaghetti query alert
Split up a complex spaghetti query into several simpler queries

reduce number of joins
Too many JOINs is a symptom of complex spaghetti queries

eliminate unnecessary distinct
Too many DISTINCT conditions is a symptom of complex spaghetti queries.

implicit column usage
Although using wildcards and unnamed columns satisfies the goal of less typing, this habit creates several hazards. This can break application refac-toring and can harm performance

having clause usage
Rewriting the query's HAVING clause into a pred-icate will enable the use of indexes during query processing.

nested sub queries
Rewriting nested queries as joins often leads to more efficient execution and more effective opti-mization

or usage
Consider using an IN predicate when querying an indexed column

union usage
Unlike UNION which removes duplicates, UNION ALL allows duplicate tuples.

distinct & join usage
The DISTINCT keyword removes duplicates after sorting the tuples. Instead, consider using a sub query with the EXISTS keyword, you can avoid having to return an entire table.

REFERENCES

[1] Poonyanuch Khumnin and Twittie Senivongse. 2017. SQL antipatterns detection and database refactoring process. 2017 18th IEEE/ACIS International Con-ference on Software Engineering, Artificial Intel-ligence, Networking and Parallel/Distributed Com-puting (SNPD)

[2] Csaba Nagy and Anthony Cleve. 2017. A Static Code Smell Detector for SQL Queries Embedded in Java Code. PReCISE Research Center, University of Na-mur, Belgium

[3] Natalia Arzamasova, Martin Schler, and Klemens Bhm. 2018. Cleaning Antipatterns in an SQL Query Log. IEEE Transactions on Knowledge and Data Engineering (Volume: 30 , Issue: 3)

[4] Bill Karwin 2010. SQL Antipatterns Avoiding the Pitfalls of Database Programming. The Pragmatic Bookshelf

[5] William J. Brown, Raphael C. Malveau, Hays W. Mc- Cormick III, and Thomas J. Mowbray. 1998. An-tipatterns. Wiley and Sons, Inc., New York.

[6] Peter Gulutzan and Trudy Pelzer 2003. SQL Performance Tuning. Addison-Wesley

[7] Vadim Tropashko 2006. SQL Design Patterns. Ram-pant Techpress, Kittrell, NC, USA

[8] J. Akbarnejad et al. 2010 SQL QueRIE recommenda-tions VLDB Endowment, vol. 3, no. 2

[9] F. Silvestri 2010 Mining query logs: Turning search usage data into knoweledge Found. Trends Inf, Retr., Vols. 4(1-2)

[10] H. Cao 2006 Context-Aware Query Suggestion by Mining ClickThrough KDD

[11] V. Singh et al. 2006 SkyServer Traffic ReportThe First Five Years Microsoft Research

[12] M. Jordan Raddick et al. 2014 Ten Years of SkyServer Tracking Web and SQL e-Science Usage Comput-ing in Science and Engineering, vol. 16(4)

[13] QFix: Di- agnosing errors through query histories eprint arXiv:1601.07539

[14] S. Brass et al. 2006 Semantic errors in SQL queries: A quite complete list Journal of Systems and Software, vol. 79, no. 5

[15] D. Burleson, V. Tropashko 2007 SQL Design Patterns: Expert Guide to SQL Programming Rampant Tech- Press, 2007