Profiling of SQL Code Snippets

1. Introduction

Software systems often need to facilitate the manipulation of data. This is commonly achieved by using the language SQL (or a variation of it). Despite the rise of NoSQL, SQL based databases remain at the forefront of data storage and there have even been instances of NoSQL DBMS's supporting SQL or SQL-like languages [5]. SQL code is a vital part of persistent data storage and poorly written SQL code can be the root of performance issues in applications [6]. Therefore, SQL tasks are prevalent in software development, and it would be beneficial to gain an empirical understanding of the difficulty of SQL related tasks.

Past research has indicated that SQL related tasks tend to take more time to complete and require different dimensions of effort than other programming tasks. A common theme in SQL related tasks is that they tend to have an increased spread of modifications [1] compared to non-SQL programming tasks.

We know that programmers tend to struggle more with SQL code. However, there is little empirical understanding of exactly why this is the case. To gain an understanding of this, our study will perform a quantitative and qualitative analysis of the quality of SQL code within open-source Java projects.

We believe that assessing the quality of SQL code snippets will aid in our understanding of why developers tend to find SQL related tasks more difficult than other tasks.

2. Research Questions

To assess the quality of SQL snippets we will draw from past research which analyzed the quality of java snippets from Stack Overflow [3]. This study defined four dimensions of snippet quality. These dimensions were derived from dimensions of software quality used in past studies [4] and adapted so that they could be applied to SQL code.

As in the study which assessed the quality of Stack Overflow code quality, the four dimensions of code snippet quality will each form a research question which overall will inform our interpretation of the code's overall quality. Thus, we have:

- **Overarching Question:** What is the quality of SQL related snippets in open-source projects?
- **RQ1:** What is the *reliability and conformance to programming rules* of SQL code snippets in OS projects?

- **RQ2:** What is the *readability* of SQL code snippets in OS projects?
- **RQ3:** What is the performance SQL code snippets in OS projects?
- **RQ4:** What is the level of *security* of SQL code snippets in OS projects?

As stated above, our primary goal with this study is to gain an empirical understanding of the quality of SQL code in open-source java projects. We believe that in doing so, we might gain insight into why programmers tend to struggle more with SQL-related tasks. Before continuing, it is important to justify why these dimensions of code quality were chosen. We further define each dimension below by adapting what was set out in past research about stack overflow code snippets [3].

(RQ1 – Reliability and conformance to programming rules): SQL Code should not be confusing or error prone, it shouldn't contain any bug or error. SQL code should adhere to generally accepted programming rules and widely accepted conventions.

(RQ2 – Readability): SQL code should follow readability conventions to ensure that code is easily understood and is able to be maintained.

(RQ3 – Performance): SQL code should consider how its formulation will affect performance and efficiency. This is especially important as database interaction, if done poorly can form the root of performance issues in applications [6].

(RQ4 – Security): SQL code should consider security constraints so as not to compromise security. This is especially important due to the nature of SQL and its ability to access sensitive information in databases.

Each of these dimensions focuses on the different consideration that a programmer writing SQL might have. These four dimensions are (for the most part) independent of each other. For example, any issues with reliability are unlikely to have serious security or readability implications and so on. Th

3. Methodology

3.1 Project Selection

We use the same 20 Apache projects used in a past study [1] which analyzed the difficulty of SQL related tasks compared to other tasks. This study has already found evidence of SQL tasks having increased difficulty, so it would be useful to further investigate the SQL code. For instance, the study found that SQL tasks tended to take longer to complete than non-SQL related tasks This is especially important since the software projects are variable in terms of style and domain [1] so if we were to identify patterns in the way SQL code is written, it could help explain this general trend.

3.2 SQL Quality Assessment

Each SQL String will be assessed against our respective static analysis tools for SQL (henceforth referred to as *SATools* which is short for *static analysis tool)*. Each tool assesses a piece of code for a set of violations and these violations are aligned with our dimensions of code quality. This will be our primary measure of assessing our four dimensions of code quality and hence, answering our research questions. A summary of the tools used and the number of checks they have is given in appendix A:

APPENDIX A: STATIC ANALYSIS TOOLS

SQL	SQLint [7]	Reliability	9 checks
	SQLFluff [8]	Readability	59 checks
	SQLCheck [9]	Performance	21checks
		Security	7 checks

4. Data Collection

4.1 Data Collection Pipeline

We want to obtain a record of all SQL code in our chosen projects. Our data extraction code will analyze the commit log of each project, and each time SQL code is detected in a file, an entry will be made in the database with the attributes as set out in appendix B.

Many of the attributes are self-explanatory, however some should be clarified. The a*dditions* and *deletions* attributes relate to the commit patch. Specifically, these attributes represent the number of lines of code that were added/deleted in that commit (indicated by a '+' or '-' character at the start of the line). The *SQL Context* and *SQL Update* attributes also relate to the commit patch. If any of the SQL code in that patch was part of the added code (was on a line starting with '+' or '-'), then the SQL code was deemed to be part of the update and the *SQL Update* attribute was flagged as true, otherwise, the SQL was deemed to be part of the code context in the patch and so the SQL Context attribute was flagged as true. To further clarify, an SQL snippet was either classed as part of the context or the update – never both and never neither.

The commit log text file (which will form as the input for our data extraction code) was created by running the following command inside the project repository directory:

git log –p > [PROJECT_NAME]-commit.txt

This command outputs a text file containing the commit log including the commit patch. This means that any code modifications as well as other commit metadata is included. Further details about the data extraction logic are outlined in section 4.3.

APPENDIX B: ATTRIBUTES INCLUDED IN SQL-DATA TABLE

- Project Name
- Commit ID (Hash)
- Author
- Date
- Time
- File Name
- Additions – Number of additions in the file containing SQL
- Deletions – Number of deletions in the file containing SQL
- SQL – The raw SQL string in this commit
- SQL Context – Flag indicating whether the SQL was included in the commit context
- SQL Update – Flag indicating whether the SQL was included in the commit diff

4.2 Project Selection

Past research also analysed open-source Java projects from Apache to gain insights about SQL [1]. We will use the same 20 projects here, however; we faced some limitations. In the end, 12 out of the possible 20 projects were used. The other 8 were excluded for the following reasons:

- ☐ Harmony, Tuscany, and OpenWebBeans had < 10 SQL entries and would therefore not be useful to compare across projects.

- ☐ ManifoldCF structured their SQL statements unconventionally, so the SQL was not accurately parsed by my routine. For example, one SQL statement from ManifoldCF is given below:

```
SELECT id AS $(IDCOLUMN), CONCAT("http://my.base.url/show.html?record=", id)  AS $(URLCOLUMN),
        CONCAT(name, " ", description, " ", what_ever) AS $(DATACOLUMN)
        FROM accounts WHERE id IN $(IDLIST)</code></p>
       CONCAT(name, " ", description, " ", what_ever) AS $(DATACOLUMN)
       FROM accounts WHERE id IN $(IDLIST)
```

  This code would not be easily parsed by our SATools and would cloud our results. Since much of the SQL scraped from this project was like the example above, it was decided that the whole project should be excluded.

- ☐ Flink, Geode, Cloudstack and Hive had commit log files too large to be processed by the Java Scanner class.

Having excluded these 8 projects, our dataset includes code from the Apache projects outlined in appendix C. This table also includes a summary of what type of project it is.

APPENDIX C: APACHE PROJECTS INCLUDED IN DATASET & PROJECT TYPE:

Project	Project Type
AsterixDB	Big Data Management System
Groovy	Programming Language
Hadoop Map-Reduce	Big Data Analysis Framework
Karaf	Container for OSGi (Open Services Gateway Initiative) frameworks
Knox	A security gateway for Apache Hadoop Map-Reduce
Lucene-Solr	Java Library for Indexing and Searching Text
Marmotta	Linked Data Server
Nifi	Data integration and data flow automation platform
OODT	Data management system framework
Oozie	Workflow scheduler system for Hadoop Map-Reduce
Synapse	Enterprise Service Bus
TomEE	Platform for deploying JavaEE applications

Total: 12

4.3 Data Collection Pipeline

4.3.1 Parsing of Commits

The first step of the data collection pipeline is to extract the commits of interest to us. Most of the commits in the text file will not contain SQL and hence be of no interest. Therefore, the first step is to filter out the commits that don't contain SQL. This was done using a regular expression which detects the beginning of SQL code. Each commit will be matched to this regular expression and omitted is not matches were found. For the remaining commits, the commit metadata is extracted, and the remaining information is sent to the next stage.

4.3.2 Retrieving the SQL code

The rest of the patch is split into individual files. Each one is checked for the presence of SQL using the same regular expression as used before. If a file contains no SQL statements it is ignored otherwise the file information is extracted along with all the SQL code in that file (this process is further detailed in subsection *4.3.3*). This information is further analysed for more information and eventually paired with the commit metadata obtained in the previous stage. This is all bundled together to form an individual entry in the CSV file.

4.3.3 SQL Code Extraction Logic

As previously mentioned, the beginning of an SQL statement is identified using the regular expression. When SQL code is identified, all the following code up until a semicolon character (indicating the end of a statement) is taken. This means that if the SQL statement is missing a semicolon, then we will get other non-SQL code in our output however this didn't seem to happen often. Depending on the context of the SQL statement, there was a small amount of treatment to the strings. For example, if the SQL statement was declared as a String e.g.

*String sql = "SELECT * FROM Users";*

then the quotation marks were removed. Furthermore, if the statement was given as an argument in a method e.g.

*conn.execute("SELECT * FROM Users");*

Then the brackets and quotation marks were removed, in both scenarios the extracted SQL statement would be:

*SELECT * FROM Users;*

4.3.4 Validations

To ensure that our regex captured SQL code, the output file was manually inspected, and it was confirmed that for the most part, only SQL code was being captured with the exception of a few instances of rubbish. An analysis of how this rubbish affected the results follows in section 4.6.

To validate the content captured by our scraping routine, the content of the output file was to run the script for the first 10 commits of each project. If the output looked good for the first 10 commits, then the rest of the commits are likely to be of similar quality. This procedure was repeated for each project due to slight variations in how the SQL was embedded. For each project, the output when considering the first 10 commits was entirely SQL and so, the script was run on the full commit logs for each project.

Another consideration when developing the scraping routine was whether each SQL statement was being captured in its entirety. If it wasn't, then our analysis of the code had the potential to be unfair since if the SATools flagged any errors introduced by us. When developing the scraping routine for some entries we would occasionally check that all the SQL code was being captured. To illustrate, we would use the commit hash from captured data to trace where the code appeared in the raw text file. If any SQL was missed, then we could amend our code (by adding a keyword to the regex for example). Once the fix was implemented, we would re-run the script and check that the missed code was included in our results.

4.4 Static Analysis Tools

Now that we have a dataset of SQL code, the next step is to run each code snippet through three static analysis tools to test for any issues regarding our four dimensions of code quality as summarized in appendix A.

These tools are command line based and take *.sql* files as input. This meant the first step of the analysis was to get the SQL code from the database and put it into their own *.sql* file. This was achieved using a script in Java (ConnectToH2.java) that established a database connection and performed a query to extract the project name, commit hash, file name, and SQL code. Then, each of the entries returned by the query became their own *.sql* file which contained the commit hash and file name inside a comment at the top and then the SQL. Each file was named based on the project it was scraped from and what number entry it was for that project (for example: groovy-18.sql).

Now that each entry was in its own *.sql* file, it could be inputted into the tools. A python script was made for each of the three static analysis tools (SQLCheck.py, SQLFluff.py, and SQLint.py). Each script executed the SATool for each of the *.sql* files by making use of the subprocess library. The output was stored in a CSV file with the attributes outlined in appendix D.

Each of the CSV files were uploaded to the database as their own table (SQL_Check, SQL_Fluff, and SQL_Lint).

The commit hash and file name were taken from the top of the *.sql* file and form the composite primary key for the table. They are also a foreign key to the original table meaning we can later perform joins. As well as storing the raw output of the tools which contains lot of additional context such as line numbers, the sql string that was flagged and other data I also made a custom summary column which parsed the raw output and created a dictionary which mapped error codes to number of times it was flagged in that file (ToolOutputParser.py). This provided a quick and easy summary of the tools output which will allow the errors to be easily tallied.

APPENDIX D: ATTRIBUTES FOR SATOOLS (SQLINT, SQLFLUFF, AND SQLCHECK)

- Commit Hash
- File Name
- [SA-Tool] Output – The raw output of the Static Analysis Tool
- [SA-Tool] Summary – Dictionary with key value pairs of error code -> number of time error occurred in that file

4.5 Summarizing the Output of the SATools

Now that the SQL code had been run through each of the tools, the next step would be to summarize its results. This was achieved using the dictionaries that mapped the error code to the frequency the violation occurred in that file. The logic to achieve this was in a Java script (ErrorCounter.java) that obtained these summary dictionaries (Lint_Summary in the SQL_Lint table, Fluff_Summary in the SQL_FLUFF table, and CHECK_Summary in the SQL_CHECK table) via a query which joined the original data to the appropriate static analysis table. This query obtained the project name, and the summary dictionary.

Then, each of the projects had the number of violations tallied by adding each of the entries to a couple of dictionaries (Java HashMap's more specifically). The first dictionary represented the violations for that project and the second dictionary represented the violations across the dataset.

This gave us the frequency of violations per project and overall.

4.6 Evaluation of Scraped Code

In section 4.3, we discussed the sanity checks we had in place when developing the scraping routine that helped to minimize the amount of rubbish that was being scraped and to minimize the number of instances where SQL statements were partially scraped. However, these sanity checks were only implemented on a small subsection of the data, specifically, the first 10 commits containing SQL for each project. This was done under the assumption that if the scraping routine performed well for the first 10 commits in each project, then it was likely to continue performing well for the renaming commits.

However, when running the script on the full commit logs, some instances of rubbish and some instances of partial SQL statements were introduced. However, this is only a problem if the tool raises because of how the SQL code was scraped. For example, if an SQL statement was

```
SELECT * FROM (
    SELECT product_id, product_name
    FROM products WHERE price > 100
) AS sub_products;
```

But our routine only partially scraped it and the entry in the dataset was

```
SELECT * FROM ( ;
```

Then if SQLCheck raised a *SELECT* * violation this would still be correct but an *unmatched parentheses* violation from SQLint would not be.

So, to quantify the degree to which our scraping routine affected our results we picked 30 random entries, determined how many were scraped incorrectly (contained rubbish or if the full statement was not scraped), and noted the number of times a violation was raised that wouldn't have been if it were scraped correctly. Note that this is different to whether the violation given by the tools was correct, here we are just interested in the frequency of incorrectly scraped queries affecting our results. We conduct an analysis of the correctness of violation in section 4.7.

At the end of this process, we found that out of 30 entries, 22 were scraped correctly meaning 8 contained either rubbish or were partially scraped. This means we can assume that approximately 26% of our entries were tainted in some way. Across these 30 entries, there were 227 violations and only 15 of those were deemed to be present as a direct result of an incorrect scraping. Therefore, we assume that approximately 6.61% of all violations are not reflections of the quality of code but introduced by us because of the way we collected our data.

Although the rate of incorrectly scraped data was surprisingly high (estimated to be just over a quarter of the data) it seems the rate of incorrectly scraped data influencing our results is relatively low (estimated to be 6.61%). However, this figure means there is an element of uncertainty in our results, and we should treat the violation statistics with a degree of caution.

4.7 Evaluation of Static Analysis Tool

Before we can perform a quantitative analysis of the tools output, we should first verify that the violations reported by the SATools are present. To do this, we will take a random sample of 10 SQL code snippet entries in our dataset and compare the SQL to the tools output. This will give us an indication of whether the tool is correctly characterized the SQL. For example, one of the SQL snippets in this validation exercise is:

update TABLE set count = 0) where count is null;

And the output of SQLint is as follows:

{"unmatched-parentheses": 1, "TOTAL": 1}

We can see that of the single violation, it was correct since there is a missing parenthesis. Therefore, for the Reliability violations, this snippet gains a score of 1/1 (100%).

The results of the full analysis are given in appendix E. All the reliability violations were deemed to be present in the SQL and the same for the readability and performance categories apart from a few. These three categories had an average correctness comfortably above 85%. However, across the 10 snippets, only two security violations were present (which is expected since there were only a few checks for security to begin with) and one of them was deemed incorrect hence its score of 50%. However, since of the small violation sample size for that category there is little evidence to suggest that the security is less accurate than the others. Therefore, we conclude that these tools do a reasonably good job at assessing

code snippets for violations although it should be noted that they occasionally flag violations inappropriately and so there is an element of uncertainty in our results.

APPENDIX E: SATOOLS EVALUATION RESULTS

Identifier	Reliability	Readability	Performance	Security
1e84b7f9747d09dc770a87f818f0e2b113bf5226 b/src/main/groovy/sql/Sql.java	1/1	1/1	1/1	0/0
d40d0074a3968b004d479afefcb5147e945460e9 b/itests/src/test/java/org/apache/karaf/itests/features/XATest.java	2/2	1/1	2/2	1/1
f91385662a09b1014f1e1935944fb55bcb47f0a0 b/gateway-server/src/main/java/org/apache/knox/gateway/services/token/impl/TokenStateDatabase.java	0/0	5/5	0/0	0/0
129a83b198e805392279208e647c23b4659a0ee0 b/solr/core/src/test/org/apache/solr/handler/TestSQLHandler.java	1/1	0/0	1/1	0/0
757b8db60f48f5fb856d3b3e8854bdc78fbfb3d1 b/libraries/kiwi/kiwi-sparql/src/main/resources/org/apache/marmotta/kiwi/persistence/pgsql/create_fulltext_index.sql	0/0	1/3	2/2	0/0
d9bcc8b4969a924847d9fe89784c166ce10fba0e b/nifi-nar-bundles/nifi-cassandra-bundle/nifi-cassandra-distributedmapcache-service/src/main/java/org/apache/nifi/controller/cassandra/QueryUtils.java	0/0	1/1	0/0	0/0
485ef7460c80302a9fa40a4b2c74b4074a059419 b/asterix-app/src/test/resources/runtimets/queries_sqlpp/copy-to/default-namespace/default-namespace.01.ddl.sqlpp	3/3	1/1	0/0	0/1
a519585b02f37c654638ece4681928fd3abbe355 b/QuestDbQueries.java	8/8	0/0	0/1	0/0
4083e074476cd64d73094fd3720c0dbff20d8f6d b/nifi-nar-bundles/nifi-iceberg-bundle/nifi-iceberg-processors/src/test/resources/hive-schema-4.0.0-alpha-2.derby.sql	7/7	1/1	1/1	0/0
131bf9ab575a6c5f0bcc05eebb75fdbf0c2abf1c b/commons/src/main/java/gov/nasa/jpl/eda/activity/MySQLStorage.java	0/0	11/12	3/3	0/0
Total	**22/22 (100%)**	**22/25 (88%)**	**10/11 (90.9%)**	**1/2 (50%)**

5. Results

5.1 Initial Insights

Before we explore the output of the tools, we already gain some insights from the raw data we collected. For example, we can see the number of SQL entries per project in appendix F. We see that AsterixDB, Nifi and Oozie have the most SQL entries. These three projects focus on data management which explains their high volume of SQL statements. Karaf, Knox and Synapse focus on providing infrastructure to manage distributed systems and have far less SQL statements. We can see that there tends to be more additions to a file than deletions and we also notice that the number of modifications to a file (additions and deletions) tends to vary from as little as ~50 lines of code to as much as ~700. There doesn't appear to be a relationship between the average number of modifications and the number of files containing SQL.

Another thing to note is that there tends to be a lot of different contributors as shown in the number of unique authors column. This indicates that the writing of SQL statements tends to be shared amongst a team. This is useful as it indicates that any repeated issues in our results are likely to have come from a wide variety of authors instead of coming from one or two authors making the same mistake. Therefore, our results will likely be more applicable to the population of programmers writing SQL code.

The next sections address each of the four code quality dimensions and research questions and the full results are detailed in appendixes G through J.

APPENDIX F: INITIAL INSIGHTS FROM SQL DATA IN EACH PROJECT

Project	No. Files containing SQL	Avg Number of additions	Avg Number of deletions	No. Unique Authors
AsterixDB	837	42	4	44
Groovy	39	149	70	7
Hadoop Map-Reduce	26	255	119	6
Karaf	23	82	23	8
Knox	23	378	321	4
Lucene-Solr	66	160	100	30
Marmotta	61	69	47	4
Nifi	218	112	77	53
OODT	54	183	116	8
Oozie	190	73	20	24
Synapse	26	71	48	8

TomEE	185	76	44	17

Number of entries: 1748

5.2 Reliability and Conformance to Programming Rules: SQLint

For reliability there were a total of 2364 violations across 1748 snippets which is approximately 1.35 violations per SQL entries

The most prominent violation for the reliability and conformance to programming rules dimension is the Invalid-Create-Option. This violation was flagged 2364 times and made up about 85% of all the violations in this category. Something to note is that most of these violations occurred in the AsterixDB project so even though it was the most common violation, this statistic is likely more representative of the AsterixDB project than the whole dataset. Upon manual inspection of the AsterixDB commit log file, we see that the SQL statements are stored in files with a *.sqlpp* file extension and statements such as:

drop dataverse test if exists;

are commonplace. This information indicates that AsterixDB is using an extension of SQL which includes additional keywords such as *DATAVERSE* and *EXTERNAL* that aren't traditional SQL. Therefore, our tool correctly validates that this code does not conform to standard SQL however, this is unlikely to indicate significant issues with the code.

The next most common violations are *unmatched parentheses* and *missing where*. Of these two violations, both are of concern. The *unmatched parentheses* violation (which makes up around 10% of all violations in this category) means the code is potentially inexecutable. Furthermore, the *missing where* violation (which makes up around 4% of all the violations in this category) refers to DELETE statements with no WHERE clause which is harmful as it has the potential to delete all the rows from a table and cause data to be lost.

TABLES 1 TO 4

RELIABILITY VIOLATIONS SUMMARY

Project	Total Violations	MCV	Freq of MCV	% MCV
AsterixDB	2050	Invalid-Create-Option	2003	97.7
Groovy	4	Unmatched-Parentheses	4	100
Hadoop Map-Reduce	4	Unmatched-Parentheses	4	100
Karaf	22	Unmatched-Parentheses	20	90.1
Knox	17	Missing-Where	17	100
Lucene-Solr	3	Unmatched-Parentheses	3	100
Marmotta	10	Unmatched-Parentheses	7	70
Nifi	166	Unmatched-Parentheses	134	80.7
OODT	38	Unmatched-Parentheses	25	73.7
Oozie	7	Missing-Where	3	42.9
Synapse	4	Unmatched-Parentheses	4	100
TomEE	39	Missing-Where	33	84.6
TOTAL	**2364**	**Invalid-Create-Option**	**2005**	**84.8**

Number of checks: 9

Number of non-offending files: 902 (51.68% Passed)

TOP FIVE MOST COMMON VIOLATIONS:

Error	Frequency	% Violation
Invalid-Create-Option	2005	84.8
Unmatched-Parentheses	240	10.2
Missing-Where	100	4.23
Trailing-Whitespace	19	0.80
-	-	-

Most Common Violations & Definition

Violation	Definition
Invalid-Create-Option	CREATE statement is given an invalid option.
Unmatched-Parentheses	Violation is raised if the number of parentheses is unmatched.
Missing-Where	Violation is raised if a DELETE statement doesn't contain a WHERE clause.
Trailing-Whitespace	Additional whitespace in the statement.

Most Common Violations & Example Offending SQL From Data:

Violation in Green

Violation	Offending SQL Example
Invalid-Create-Option	**Asterixdb:** CREATE EXTERNAL COLLECTION DeltalakeDataset1(DeltalakeTableType) USING %adapter% (%template%, ("container"="playground"), ("definition"="delta-data/delta_all_type"), ("decimal-to-double" = "true"), ("timestamp-to-long" = "false"), ("date-to-int" = "false"), ("timezone" = "PST"), ("table-format" = "delta"));
Unmatched-Parentheses	**Groovy:** update TABLE set count = 0) where count is null;
Missing-Where	**OODT:** DELETE FROM workflow_instances;
Trailing-Whitespace	**OODT:** CREATE TABLE foo_metadata (product_id int NOT NULL, element_id varchar(1000) NOT NULL, metadata_value varchar(2500) NOT NULL);

5.3 Readability: SQL-Fluff

For readability, there were a total of 8767 violations across 1748 entries which is approximately 5 violations per SQL entry.

From these results we can see that these violations are very prevalent since only 10 entries passed (had no violations) out of the 1748 total. This is somewhat surprising as a lot of the queries in the dataset are relatively short and straightforward. However, this SATool has a lot of checks (nearly 60) and some of the checks for violations are easily triggered. An example of a query that had no violations is:

```
INSERT INTO hadoop_output VALUES (?,?,?);
```

Interestingly, approximately 47% of queries (827 out of 1748) just had a single flag for LT05 (excessively long queries with no line break) one such query is:

```
CREATE TYPE nodetype AS ENUM ("uri","bnode","string","int","double","date","boolean");
```

Since nearly half of all queries had just this violation but otherwise passed it could be interpreted that aside from queries not tending to include line breaks (which is expected in the context of embedding queries in Java code) the readability of SQL code tends to be okay about half the time.

The most prevalent violation was CP02 which was raised when there were capitalization inconsistencies. Capitalization is very important for the readability of SQL queries since it is used to indicate SQL keywords which readers break down the queries logic. However, since SQL is case insensitive, it makes sense that this convention is not widely adopted and hence why we see a large proportion of queries (about 25%) being flagged with this issue.

Explanation for omitted violations

Two violations were omitted from the results. These violations were 'PRS' which was raised if any part of the SQL was unable to be parsed and 'LT12' which was raised if the SQL text didn't end with a newline character. These were omitted since PRS tended to be raised for unconventional keywords for example one AsterixDB query: CREATE DATAVERSE test; resulted in a PRS error since the SATool (SQLFluff) was configured for MySQL which doesn't include the keyword DATAVERSE. Additionally, LT12 was omitted since missing a newline character is an issue with the way the data has been scraped – not an issue with the SQL content. Furthermore, most of the SQL was embedded in Java code and this violation relates to the convention for *.sql* files.

TABLES 4 TO 8

READABILITY VIOLATIONS SUMMARY

Project	Total Violations	MCV	Freq of MCV	% MCV
AsterixDB	1334	LT05	844	63.3
Groovy	238	AM04	61	25.6

Hadoop Map-Reduce	87	LT05	22	25.3
Karaf	52	LT05	23	44.2
Knox	48	LT05	28	58.3
Lucene-Solr	433	LT14	138	31.8
Marmotta	374	LT01	191	51.1
Nifi	1180	CP02	358	30.3
OODT	296	LT01 & LT02	56	18.9
Oozie	2911	CP02	1005	34.5
Synapse	90	LXR	50	55.6
TomEE	1724	CP02	616	35.7
TOTAL	**8767**	**CP02**	**2229**	**25.4**

Number of Checks: 57 (2 Omitted)

Number of non-offending files: 10 (0.57% Passed)

Top Five Most Common Violations:

Error	Frequency	% Violation
CP02	2229	25.4
LT05	2003	22.8
LT01	862	9.83
LT14	793	9.04
AL01	572	6.52

Most Common Violations & Definition

Violation	Definition
CP02	Capitalization inconsistencies can make debugging difficult as it is hard to differentiate SQL keywords from other text in the query.
LT05	Queries are excessively long with no line break introducing unnecessary confusion.
LT01	Inappropriate spacing such as excessive or trailing whitespace.
LT14	Keywords should be before/ after a newline.
AL01	Tables are aliased without AS keyword.

Most Common Violations (MCV) & Example Offending SQL From Data:

Violation	Offending SQL Example
CP02	**Groovy:** drop table if exists PERSON; DROP TABLE IF EXISTS person;
LT05	**Karaf:**

	CREATE TABLE messages (id INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY, message VARCHAR(1024) NOT NULL, CONSTRAINT primary_key PRIMARY KEY (id))"));
LT01	**Lucene-Solr:** SELECT C.COUNTRY_NAME FROM PEOPLE P';
LT14	**Groovy:** select * from PERSON where firstname like "S%";
AL01	**Nifi:** select U.active from users U; (no AS)

5.4 Performance: SQL-Check

For performance, there were a total of 1034 violations across 1748 SQL entries which is approximately 0.59 violations per entry.

The most prevalent violations were *index attribute order* and *SELECT* * (which made up about 37% and 29% of all the violations respectively). The first violation refers to when an index is created that goes against the underlying order of the DBMS. This means that each query becomes more computationally expensive because there is additional logic needed to order the results. The other violation is *SELECT* * refers to the fact that it's likely that not all columns will be useful and so it is more resource efficient to not query all the columns. Using the * wildcard can also be a security risk as outlined in the next section. One thing to note regarding this code quality dimension is the spread of total violations meaning that the results are not dominated by any violation. Also, with a large volume of flagged queries (~37%), it indicates that for the performance of SQL code, there are lots of commonplace issues for programmers to be aware of.

TABLES 8 TO 12

PERFORMANCE VIOLATIONS SUMMARY

Project	Total Violations	MCV	Freq of MCV	% MCV
AsterixDB	370	Index Attribute Order	267	72.2
Groovy	79	SELECT *	78	98.7
Hadoop Map-Reduce	2	Metadata Tribbles & Implicit Column Usage	1	50
Karaf	3	Readable Passwords	2	66.7
Knox	10	SELECT *	8	80
Lucene-Solr	98	SELECT *	88	90
Marmotta	96	Index Attribute Order	66	69
Nifi	200	SELECT *	67	34
OODT	47	Implicit Column Usage	29	61.7

Oozie	64	Index Attribute Order	36	56.3
Synapse	42	Implicit Columns Usage	40	95.2
TomEE	23	SELECT *	11	47.8
TOTAL	**1034**	**Index Attribute Order**	**377**	**36.5**

Number of Checks: **14**

Number of non-offending files: 1095 **(62.6% Passed)**

Top Five Most Common Violations:

Error	Frequency	% Violation
Index Attribute Order	377	36.5
SELECT *	299	28.9
Implicit Column Usage	116	11.2
Metadata Tribbles	81	7.83
Generic Primary Key	60	5.80

Most Common Violations & Definition

Violation	Definition
Index Attribute Order	The query attributes are not in the same order as the index attributes. This means the query goes against the DBMS's natural indexing resulting in a more computationally expensive query.
SELECT *	Query all columns in a table when it is unlikely that they are all needed.
Implicit Column Usage	Like SELECT *, this violation refers to wildcard usage and unnamed columns in querying.
Metadata Tribbles	Storing the same entity over multiple columns, for a column for each year representing revenue (e.g. revenue2002, revenue2003…), this results in a larger volume of data which degrades performance.
Generic PK	Having a primary key that is unrelated to the data (e.g., id). This makes joining tables more computationally expensive since the query becomes more complex.

Most Common Violations & Example Offending SQL From Data:

Violation	Offending SQL Example
Index Attribute Order	**Marmotta:** CREATE INDEX idx_triples_spo ON triples(subject,predicate,object) WHERE deleted = false;
SELECT *	**Tomee:** select * from entity where id = ?;
Implicit Column Usage	**Hadoop Map-Reduce:** INSERT INTO hadoop_output VALUES (?,?,?);
Metadata Tribbles	**Marmotta:**

	CREATE INDEX idx_versions_added ON versions_added(version_id);
Generic PK	**Karaf:** CREATE TABLE messages (id INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY, message VARCHAR(1024) NOT NULL, CONSTRAINT primary_key PRIMARY KEY (id))"));

5.5 Security: SQL-Check

For security, there were a total of 334 violations across the 1748 SQL entries which is approximately 0.19 violations per snippet.

In terms of the security of SQL code snippets, the most prevalent violation by far was SELECT *. This violation made up nearly 90% of all the security violations and this violation appeared nearly 300 times across all 1748 code snippets in the dataset. As well as the performance issues detailed in the previous section, using the * wildcard to retrieve all columns in a table poses a security threat because it increases the likelihood of sensitive information being compromised. Additionally, if the table structure changes (new columns added) then queries might expose unintended data, further contributing to the risk of data leakage. Aside from *SELECT *,* the other two violations that appeared were using a data type (float) that is susceptible to precision errors, and having passwords stored in plain text. Interestingly, all three of these security violations are easily fixed (query only necessary columns, use double instead of float and hash passwords before storing). This indicates that for these violations, it is important that programmers are aware of these antipatterns so they can be easily avoided.

Explanation for omitted violations: Two violations *spaghetti query* and *string concatenation* were omitted from the results. The spaghetti query violation was raised if a query would result in a cartesian product of two or more tables. This would pose a huge security risk as it would expose a significant amount of potentially sensitive data if it were obtained by a malicious third party. The s*tring concatenation* violation was raised if concatenation was used to formulate the query which would make the code susceptible to SQL injection. However, of the 49 times the spaghetti query violation was raised and the singular time the string concatenation query was raised, there was little evidence of the flagged query violating its respective rule.

TABLES 12 TO 16

SECURITY VIOLATIONS SUMMARY

Project	Total Violations	MCV	Freq of MCV	% MCV
AsterixDB	56	SELECT *	32	57.1
Groovy	78	SELECT *	78	100
Hadoop Map-Reduce	0	-	0	0
Karaf	3	Readable Passwords	2	66.7

Knox	8	SELECT *	8	100
Lucene-Solr	88	SELECT *	88	100
Marmotta	1	Imprecise Data Type	1	100
Nifi	72	SELECT *	67	93.1
OODT	14	SELECT *	14	100
Oozie	0	-	0	0
Synapse	0	-	0	0
TomEE	14	SELECT *	11	78.6
TOTAL	**334**	**SELECT ***	**299**	**89.5**

Number of Checks: 5 (2 Omitted)

Number of non-offending files: 1552 (88.8% Passed)

Top Three Most Common Security Violations:

Violation	Frequency	% Violation
SELECT *	299	89.5
Imprecise Data Type	30	8.98
Readable Passwords	5	1.50

Most Common Security Violations & Definition

Violation	Definition
SELECT *	Querying with a wildcard means all the columns are returned, this increases the risk of sensitive columns being accessed by a third party.
Imprecise Data Types	Using *float* datatype as opposed to *double* and *real*. This is because using *float* can lead to precision errors when being inserted into the database.
Readable Passwords	Passwords are stored in plain text.

Most Common Security Violations & Example Offending SQL From Data:

Violation	Offending SQL Example
SELECT *	**Tomee:** select * from entity where id = ?;

Imprecise Data Types	**Nifi:** create table mytable (id integer not null, name varchar(100), scale float, created timestamp, data blob)
Readable Passwords	**Tomee:** CREATE TABLE users (user VARCHAR(255), password VARCHAR(255))

APPENDIX G: RELIABILITY FULL RESULTS

PROJECT_NAME	missing-where	invalid-create-option	trailing-whitespace	unmatched-parentheses
asterixdb	11	2003	5	31
groovy				4
hadoop-mapreduce				4
karaf	1		1	20
knox	17			
lucene-solr				3
marmotta	1	2		7
nifi	31		1	134
oodt	3		10	25
oozie	3		2	2
synapse				4
tomee	33			6
Total	**100**	**2005**	**19**	**240**

APPENDIX H: READABILITY FULL RESULTS

PROJECT_NAME	AL01	AL02	AL03	AL04	AL05	AM04	CP02	CP03	CV06	LT01	LT02	LT05	LT08	LT09	LT14	RF01	RF02	ST11	CP01	RF04	CV01	CP05	RF03	CV10	RF05	RF06	Total
asterixdb	24	5	2	2	15	10	68	8	5	22	23	52	3	5	17	3	4	10								278	556
groovy							1												4							5	10
hadoop-mapreduce							1					1														2	4
karaf												2							4	4						10	20
knox						8						5														13	26
lucene-solr						23	3		1	11		24		4	62						3					131	262
marmotta							23			103	36	34		4	6				1	4		5				216	432
nifi	6					52	263		11	26		109		5	38		2	4	47	72			5			640	1280
oodt	3				3	6	34		10	36	37	22		5	19				6					6	3	190	380
oozie	30				25		34	4	2	10		29			5		1		8				8			156	312
synapse																										0	0
tomee	24				65	2	352			272		120			1				87	19		2	2			946	1892
Total	**87**	**5**	**2**	**2**	**108**	**101**	**779**	**12**	**29**	**480**	**96**	**398**	**3**	**23**	**148**	**3**	**7**	**14**	**157**	**99**	**3**	**7**	**15**	**6**	**3**	**2587**	**5174**

APPENDIX I: PERFORMANCE FULL RESULTS

PROJECT_NAME	GROUP BY Usage	Implicit Column Usage	Index Attribute Order	SELECT *	UNION Usage	Metadata Tribbles	Generic Primary Key	Recursive Dependency	Files Are Not SQL Data Types	Multi-Valued Attribute	Total
asterixdb	13	2	267	32	11						325
groovy				78							78
hadoop-mapreduce		1				1					2
karaf				1			1				2
knox				8		2					10
lucene-solr	8		2	88							98
marmotta		23	66			6		1			96
nifi		21	3	67		54	54				199
oodt		29	3	14		1					47
oozie			36			10			14	4	64
synapse		40								1	41
tomee				11		7	5				23
Total	**21**	**116**	**377**	**299**	**11**	**81**	**60**	**1**	**14**	**5**	**985**

APPENDIX J: SECURITY FULL RESULTS

PROJECT_NAME	Imprecise Data Type	SELECT *	String Concatenation	Readable Passwords	Total
asterixdb	24	32	1		57
groovy		78			78
hadoop-mapreduce					0
karaf		1		2	3
knox		8			8
lucene-solr		88			88
marmotta	1				1
nifi	5	67			72
oodt		14			14
oozie					0
synapse					0
tomee		11		3	14
Total	**30**	**299**	**1**	**5**	**335**

6. Summary

This study is intended to follow up from the results of previous work which found that programmers tend to find SQL related tasks more difficult than non-SQL related tasks [1]. This is indicated by there being a significant difference in their time to completion and evidence of SQL tasks requiring different dimensions of effort compared to non-SQL related tasks. Our study aims to provide context on the overall quality of SQL code in the same open-source projects as focused on in the past study. We hoped that by gaining an empirical understanding of the quality of SQL code, it would provide insight into why programmers tend to struggle with SQL related tasks.

Our analysis defined code quality in terms of four dimensions and used static analysis tools to assess each of them. From this, we identified some prevalent antipatterns in SQL code and gained a sense of what dimensions the SQL code performed well and not so well in. For example, there were an average of 5 violations per snippet in the readability category and an average of 1.35 violations per snippet in the reliability category. However, for reliability we might consider excluding the ~2000 *invalid create statement* violations since they mostly came from AsterixDB and deemed not to be an antipattern in most cases. Excluding this violation, the average number of violations per SQL entry becomes 0.2. For readability, we notice that this category has more checks than the other code quality dimensions, however the high number of violations to code snippets is still sufficient evidence to conclude that SQL code in open-source projects can tend to suffer from poor readability.

In terms of performance and security, there was on average 0.59 and 0.19 violations per SQL entry respectively. Whilst the code performed better in these two categories than the others, this is still of concern considering that performance and security violations might have greater implications for the system. Good database performance is of huge importance for any application and huge amounts of time and effort are invested to find ways to save space and time for users. Furthermore, a single security vulnerability if exploited can result in data breaches or expose underlying information about the system that can be used for further exploitation. So even though there were less occurrences of violations in these categories, the implications of these violations are massive.

Overall evidence this project points to the quality of SQL code being questionable particularly in terms of readability although there are certainly areas to improve in the reliability, performance and security of the code. However, it is also important to note there is some uncertainty in our results due to our scraping routine being unfair in some cases and some false positives in violations also detract from the validity of the static analysis tools.

Having gained this understanding around the quality of SQL code in open-source projects, it helps inform why previous research found evidence of developers struggling with it more than non-SQL code. Since we know that SQL code is susceptible to poor readability, it makes sense that developers might struggle to work on tasks that involve it. Additionally, the prevalence of antipatterns across many unique authors further reinforces the idea that developers might not engage as much with writing SQL as they do other languages or programming tasks. Coupling this with the occasional antipattern relating to reliability, performance, and security, we have a new level of insight into why developers tend to struggle more when faced with SQL related tasks compared to other tasks.

7. References

[1]: Studying the characteristics of SQL-related development tasks: Costa et al. 2023

[2]: Code Reuse in Stack Overflow and Popular Open Source Java Projects: Licorish et al. 2018

[3]: Understanding stack overflow code quality: Meldrum et al., 2020

[4]: A.L. Ginsca, A. Popescu, User profiling for answer quality assessment in Q&A communities, in: Proceedings of the 2013 Worksho p on Data-Driven User Behavioral Modelling and Mining from Social Media, ACM, 2013, pp. 25–28.

[5]: Gaspar D. Codric I (2017) Bridging relational and NoSQL databases

[6]: Faroult S, Robson P (2006) The Art of SQL

[7]: Brass, S., & Goldberg, C. (2004, 8-9 Sept. 2004). Semantic errors in SQL queries: a quite complete list. Paper presented at the Fourth International Conference on Quality Software, 2004. QSIC 2004. Proceedings. doi:10.1109/QSIC.2004.1357967

[8]: Fluri, J., Fornari, F., & Pustulka, E. (2023, 14-15 May 2023). Measuring the Benefits of CI/CD Practices for Database Application Development. Paper presented at the 2023 IEEE/ACM International Conference on Software and System Processes (ICSSP). doi:10.1109/ICSSP59042.2023.00015

[9]: Dintyala, P., Narechania, A., & Arulraj, J. (2020). SQLCheck: Automated Detection and Diagnosis of SQL Anti-Patterns. Paper presented at the Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA. doi:10.1145/3318464.3389754