SQLCheck+: A SQL-based Database-access Performance AP Detection Tool

Weisheng Wang (ww2609)

1 Synopsis

Data storage has become one of the most common features for web applications. While software developers design database schemas and program query statements to retrieve data, they may inevitably introduce database-access performance anti-patterns (AP) which not only violate basic design principles and best practices but also bring down the overall system performance. For instance, APs may cause redundant storage and increase applications' response time. Besides, identifying APs and implementing fixes require much developers' effort and are time-consuming. To understand and handle database-access performance issues, researchers have examined popular real-world web applications to extract and categorize common performance APs and evaluated the APs' impacts on applications' performance. Some studies propose SQL level APs detection and fix suggestion approaches. With using object-relational-mapping (ORM) frameworks, which are written in object-oriented programming languages to provide data model abstraction for developers to reduce the burden of CRUD operations, to build web applications become popular, some researchers have examined ORM-based applications and also found ORM level APs that affect performance. Some studies focus on mobile applications and analyze how APs affect energy consumption and may introduce security vulnerabilities.

For my project, I developed SQLCheck+ to detect logical design and physical design related SQL-based APs. My motivation stemmed from the fact that the open-sourced SQLCheck [1] tool has a relatively high false-positive rate and is incompetent in detecting APs in many scenarios. After examining the false-positive cases from my midterm paper experiments, I found it solvable by implementing additional rule-based approaches and analyzing raw data's contents. Additionally, although data analysis is proposed in SQLCheck, the open-sourced SQLCheck tool does not support inputting raw data. Furthermore, while SQLCheck only uses data analysis to help detect APs, SQLCheck+ performs data analysis both to identify potential APs and to verify SQLCheck's reported APs to reduce the false-positive rate. After implementing SQLCheck+ and comparing it against SQLCheck over 9 popular Kaggle datasets, SQLCheck+ has shown a significantly high precision (True Positive/(True Positive + False Positive)).

1.1 AP Coverage

SQLCheck+ focuses on detecting logical design and physical design APs. According to SQLCheck, logical design AP is the practice that violates the principles of data organization and interconnectivity that will cause an increase in update and maintenance overheads. For instance, violation of primary key constraints and foreign key constraints fall in this category. Physical design AP is the practice that may inevitably introduce accuracy problems. For example, using FLOAT or other floating-point types may cause rounding errors when computing query results.

SQLCheck+ is able to detect a wide range of APs. For logical design APs, SQLCheck+ is able to detect Multi-Value Attribute, No Primary Key, Generic Primary Key, No Foreign Key, and Metadata Tribbles (also called Data In Metadata in SQLCheck paper); for physical design APs, SQLCheck+ is able to detect Imprecise Data Type and Enumerated Types APs. For more details about the APs' definition and potential performance issues, readers can refer to https://github.com/jarulraj/sqlcheck/tree/master/docs/logical and https://github.com/jarulraj/sqlcheck/tree/master/docs/physical.

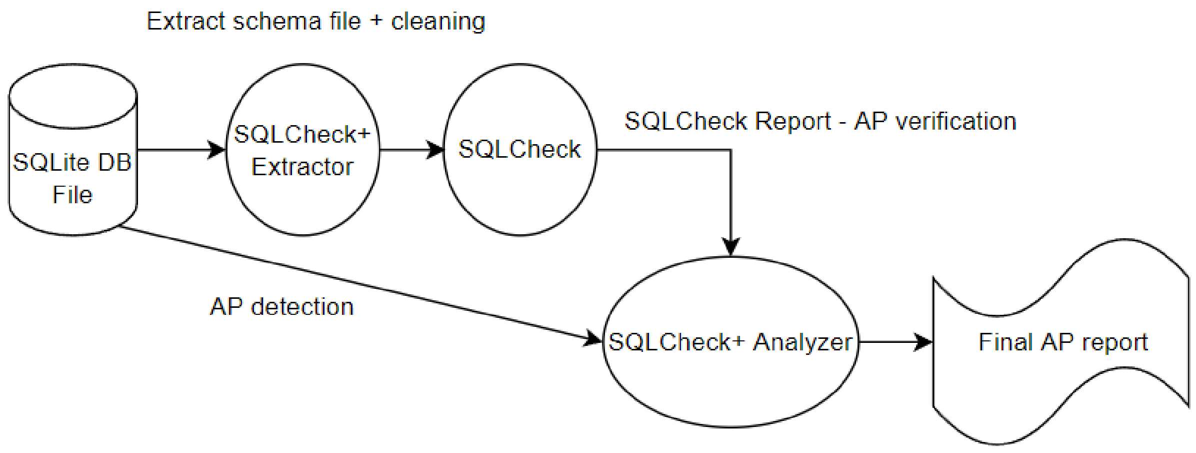


Figure 1. SQLCheck+ System Overview

1.2 System Architecture and Workflow

SQLCheck+ consists of 3 components: SQLCheck+ Extractor, SQLCheck, and SQLCheck+ Analyzer. In the beginning, the user will provide an SQLite DB file, which normally contains the database's schema information and data records, to SQLCheck+. Then, SQLCheck+ will first run the Extractor to extract the schema information from the database and store it in a local sql file. The extractor will also perform necessary cleaning such as removing quotation marks to ensure other components can safely use the schema file. Next, the SQLCheck component will run on the schema file and generate an AP report. SQLCheck+ uses the original SQLCheck for this component because the idea is to first get an AP report from SQLCheck and verify if its reported APs are correct. Then, SQLCheck+ will run the Analyzer to perform verification of the SQLCheck's report and AP detection tasks, and finally, generate the combined AP report.

SQLCheck implementation details can be found at https://github.com/jarulraj/sqlcheck.
SQLCheck+ implementation details can be found at https://github.com/BotMichael/SQLCheck-Plus.

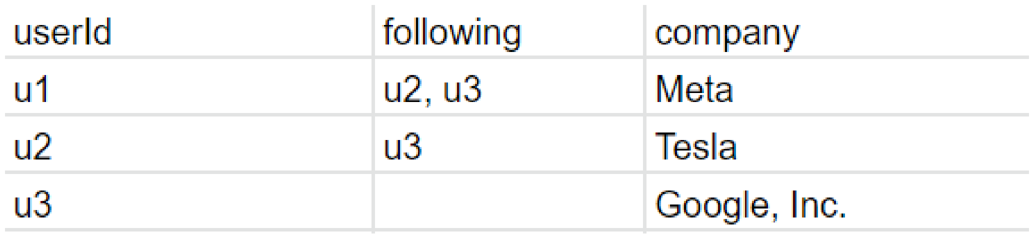
1.2.1 Sqlcheck+ Extractor

This component is responsible for extracting database schema from the SQLite DB file, and storing the schema into a local file.

1.2.2 Sqlcheck+ Analyzer

This component is responsible for two major tasks: AP detection and AP verification.

AP detection: The Analyzer has implemented additional rule-based AP detection methods and utilized data analysis algorithms to detect APs. In particular, Multi-Value Attribute AP needs data analysis to examine its existence. For Multi-Value Attribute AP, the Analyzer will look into the table's TEXT type columns and collect the frequencies of words/phrases separated by a delimiter (used ‘, ’ as the default delimiter). Then, the Analyzer will compare and find whether any word/phrase has exceeded the default threshold (the default is 0.15 and users can change the threshold). For instance, suppose there is a User table with ‘userId’, ‘following’, and ‘company’ columns. If the following column stores ‘, ’ separated userId values, then obviously, it violates the atomicity principle. In this case, having a low threshold like 0.001 would be helpful since we want to ensure that as long as the ‘following’ columns store ‘, ’ separated user ids, we can detect the Multi-Value Attribute AP. However, it’s also possible for the ‘company’ column to contain ‘, ’ separated values whereas they don’t really violate the Multi-Value Attribute AP. In this case, a relatively high threshold like 0.3 would be helpful to prevent false positives. To meet different users’ requirements, SQLCheck+ allows the users to specify the threshold value when running the tool.



userId	following	company
u1	u2, u3	Meta
u2	u3	Tesla
u3		Google, Inc.

Figure 2. Exemplary User table

AP verification: The Analyzer will verify each reported AP from the SQLCheck’s report and use the same rule-based methods and data analysis algorithms to verify the AP.

2 Research Questions

To compare SQLCheck+ with SQLCheck, we need a benchmark dataset to evaluate their performance. To my understanding, there’s no such dataset. Therefore, I collected 9 popular SQLite datasets (each with 200+ upvotes) from Kaggle, extracted the schema files using SQLCheck+ Extractor (with Python’s SQLite3 library), and inspected and manually labeled the SQL statements to answer the first research question about the distribution of logical design and physical design APs in the datasets. After getting the ground truth for the SQL statements, to study my second research question, I ran SQLCheck and SQLCheck+ on the datasets and compared their performance.

The labeled distribution and results from both tools are shown in Figure. Appendix 1. For each cell, the first value is the number of the corresponding APs detected by SQLCheck; the second is the SQLCheck+’s result; the third is the manually labeled result. For all the datasets except the Pitchfork Reviews dataset, the Multi-Value Attribute AP threshold has been set to 0.15 and for the Pitchfork Reviews dataset, the threshold has been set to 0.001.

Links to the 9 Kaggle datasets and manually labeled datasets have been added in Appendix 6.1.

2.1 RQ1: What’s the distribution of APs in the 9 popular Kaggle datasets?

As we can observe from Figure. Appendix 1, there are no Metadata Tribbles and Enumerated Types AP among the 9 datasets. And Multi-Value Attribute and Imprecise Data Type are not common as well (1 Multi-Value Attribute AP found in the *Hillary Clinton's Emails* dataset and 1 Multi-Value Attribute AP and 1 Imprecise Data Type found in the *18,393 Pitchfork Reviews* dataset).

However, there are a large number of No Primary Key, Generic Primary Key, and No Foreign Key APs detected among the 9 datasets:

For *The History of Baseball*, *18,393 Pitchfork Reviews,* and *US Consumer Finance Complaints* datasets*,* none of the tables have specified a primary key and 2 out of 3 tables of the *Mental Health in the Tech Industry* dataset also lack primary keys.

For *Hillary Clinton's Emails*, *NIPS Papers*, *SF Salaries*, and *US Baby Names* datasets, all the tables have violated the Generic Primary Key AP (using 'id' to name the tables' id column instead of following the convention '<tablename>_id').

For *Hillary Clinton's Emails, Mental Health in the Tech Industry, NIPS Papers, 18,393 Pitchfork Reviews,* and *The History of Baseball* datasets, a large proportion of the tables do not have foreign key constraints enforced.

2.2 RQ2: how well does SQLCheck+ compare with SQLCheck

	# Manually Labeled APs	SQLCheck (TP, FP, missed, coverage, precision)	SQLCheck+ (TP, FP, missed, coverage, precision)
Hillary_Clinton_s_Emails	8	(4, 2, 4, 0.5, 0.67)	(8, 0, 0, 1, 1)
Mental_Health	3	(0, 4, 3, 0, 0, 0)	(3, 0, 0, 1, 1)
NIPS_Papers	4	(3, 0, 1, 0.75, 1)	(4, 0, 0, 1, 1)
Pitchfork-Reviews	13	(1, 0, 12, 0.08, 1)	(13, 0, 0, 1, 1)
SF_Salaries	1	(1, 0, 0, 1, 1)	(1, 0, 0, 1, 1)
Twitter-US-Airline-Sentiment	0	(0, 0, 0, null, null)	(0, 0, 0, null, null)
US_Baby_Names	2	(2, 0, 0, 1, 1)	(2, 0, 0, 1, 1)
US_Consumer_Finance_Complaints	1	(0, 0, 1, 0, null)	(1, 0, 0, 1, 1)
The-History-of-Baseball	49	(0, 27, 49, 0, 0)	(49, 1, 0, 1, 0.98)

Table 1. AP reports

For RQ2, we compare the performance of SQLCheck and SQLcheck+ in terms of coverage(TP/Manually Labeled) and precision (TP/(TP+FP)). TP stands for True Positive and FP stands for False Positive. The result is shown in Table 1 and the null value is due to the denominator being 0. As we can observe from both metrics, SQLCheck+ has a steady and significantly high coverage and precision performance in all datasets, whereas, SQLCheck's performance is not stable and in some datasets, SQLCheck performs extremely bad.

The problem with SQLCheck's performance is mostly because of its heavy reliance on regular expression matching while SQLCheck+ implements more rule-based methods and leverages data analysis to help with AP detection. For instance, to detect Multi-Value Attribute AP, SQLCheck simply uses a regular expression of "id text" since it assumes id columns should not store TEXT values. However, storing encrypted ids in text type is very common nowadays and should not be blindly attributed to having Multi-Value AP. Another example is that it uses regular expressions "null" to detect the NULL Usage AP which is a query AP rather than a logical design or a physical design related AP. In addition, as discussed in section 1.2.2 about the data analysis algorithm, since Multi-Value Attribute AP may not only exist in id columns, it's also important to examine any columns that have TEXT type, which is also one of the reasons why SQLCheck missed detecting some Multi-Value Attribute APs.

3 Deliverables

The repository of the project is at https://github.com/BotMichael/SQLCheck-Plus. The preferable environment is Windows 10. Other operating systems should be fine to use but the user needs to install the corresponding SQLCheck software so that SQLCheck+ can use it for the SQLCheck component.

Sample SQLite DB files can be downloaded from the links in Appendix 6.2. The user can put an SQLite DB file in the exp folder and run the command:

```
python3 main.py ./exp/<filename>.sqlite <optional threshold value>
```

More details can be found in the project's README.md file.

4 Self-evaluation

Through the project, I've developed and implemented SQLCheck+, a SQL-based AP detection tool for detecting logical design and physical design APs. From RQ2, it has been shown that my SQLCheck+ outperformed the original SQLCheck for all 7 APs (Multi-Value Attribute, No Primary Key, Generic Primary Key, No Foreign Key, Metadata Tribbles, Imprecise Data Type, and Enumerated Types). I've also created a manually labeled dataset that can be used for future studies.

During SQLCheck+'s development, I find one of the most challenging tasks is to think of all the potential AP scenarios so that I could design a comprehensive solution to detect the APs. It's also challenging to manually label the SQL statements because some APs are quite hard to find manually which further makes me believe that developing SQLCheck+ to automate the detection process is valuable for this type of software engineering practice.

Originally, I planned to implement query AP detection for SQLCheck+ as well. However, I did not find a good dataset for this task(with SQLite file and query statements both available). So I decided to focus on detecting logical design and physical design APs. In addition, SQLCheck+ is very modular so adding new detection methods and modifying old ones is easy.

Overall, I really love this course and appreciate the invaluable opportunity to explore a database-related topic, experience writing literature reviews for a domain, and do projects and conduct experiments to verify my ideas.

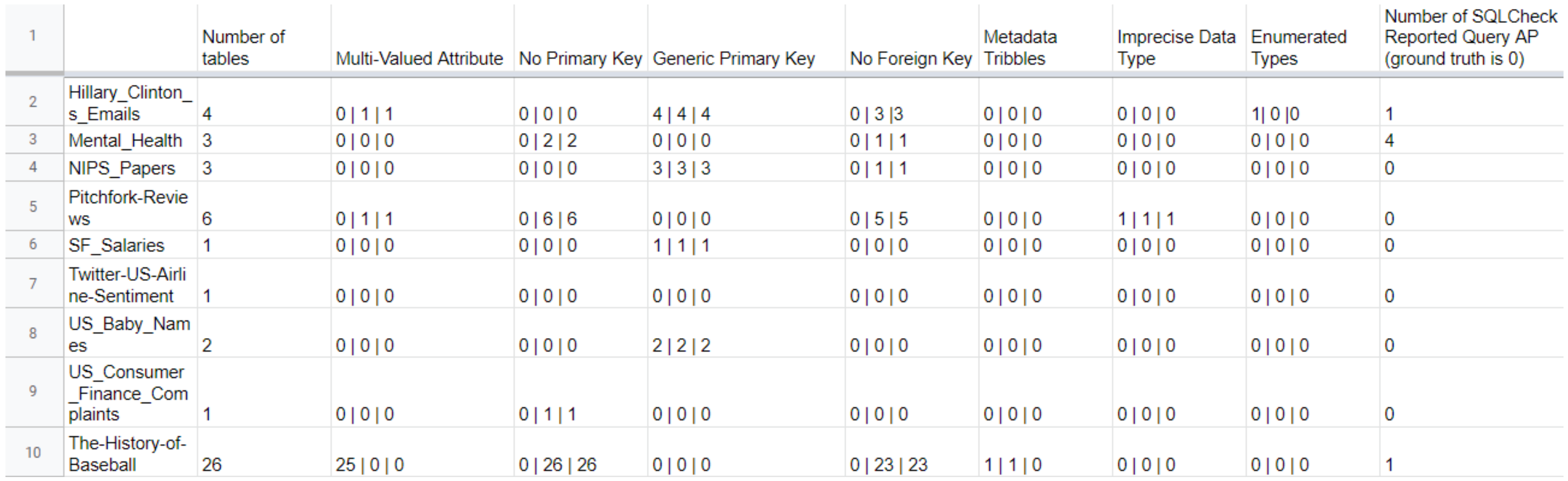
I'd like to point out that I find the scope of available papers for presentations could be more comprehensive. However, that students may request for presenting papers they find interesting helps enlarge the scope, so it's not much of a problem.

5 References

1. Dintyala, P., Narechania, A., & Arulraj, J. (2020, June). SQLCheck: automated detection and diagnosis of SQL anti-patterns. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (pp. 2331-2345). https://doi.org/10.1145/3318464.3389754

6 Appendix

6.1



1		Number of tables	Multi-Valued Attribute	No Primary Key	Generic Primary Key	No Foreign Key	Metadata Tribbles	Imprecise Data Type	Enumerated Types	Number of SQLCheck Reported Query AP (ground truth is 0)
2	Hillary_Clinton_s_Emails	4	0 | 1 | 1	0 | 0 | 0	4 | 4 | 4	0 | 3 |3	0 | 0 | 0	0 | 0 | 0	1| 0 |0	1
3	Mental_Health	3	0 | 0 | 0	0 | 2 | 2	0 | 0 | 0	0 | 1 | 1	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	4
4	NIPS_Papers	3	0 | 0 | 0	0 | 0 | 0	3 | 3 | 3	0 | 1 | 1	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0
5	Pitchfork-Reviews	6	0 | 1 | 1	0 | 6 | 6	0 | 0 | 0	0 | 5 | 5	0 | 0 | 0	1 | 1 | 1	0 | 0 | 0	0
6	SF_Salaries	1	0 | 0 | 0	0 | 0 | 0	1 | 1 | 1	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0
7	Twitter-US-Airline-Sentiment	1	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0
8	US_Baby_Names	2	0 | 0 | 0	0 | 0 | 0	2 | 2 | 2	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0
9	US_Consumer_Finance_Complaints	1	0 | 0 | 0	0 | 1 | 1	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0
10	The-History-of-Baseball	26	25 | 0 | 0	0 | 26 | 26	0 | 0 | 0	0 | 23 | 23	1 | 1 | 0	0 | 0 | 0	0 | 0 | 0	1

Figure Appendix 1. Manually labeled APs and results from SQLCheck and SQLCheck+

6.2 Dataset Links

Kaggle Datasets

18,393 Pitchfork Reviews
https://www.kaggle.com/datasets/nolanbconaway/pitchfork-data

Twitter US Airline Sentiment
https://www.kaggle.com/datasets/crowdflower/twitter-airline-sentiment

SF Salaries
https://www.kaggle.com/datasets/kaggle/sf-salaries

US Consumer Finance Complaints
https://www.kaggle.com/datasets/kaggle/us-consumer-finance-complaints

US Baby Names
https://www.kaggle.com/datasets/kaggle/us-baby-names

Mental Health in the Tech Industry
https://www.kaggle.com/datasets/anth7310/mental-health-in-the-tech-industry

NIPS Papers
https://www.kaggle.com/datasets/benhamner/nips-papers

Hillary Clinton's Emails
https://www.kaggle.com/datasets/kaggle/hillary-clinton-emails

The History of Baseball
https://www.kaggle.com/datasets/seanlahman/the-history-of-baseball

Manually Labeled Dataset
https://github.com/BotMichael/SQLCheck-Plus