Interactive Demonstration of SQLCheck

Arthita Ghosh*
Georgia Institute of Technology
Atlanta
aghosh80@gatech.edu

Deven Bansod*†
Facebook, Inc.
Menlo Park, CA
dbansod@gatech.edu

Arpit Narechania
Georgia Institute of Technology
Atlanta
arpitnarechania@gatech.edu

Prashanth Dintyala†
NVIDIA Corporation
vdintyala3@gatech.edu

Su Timurturkan
Georgia Institute of Technology
Atlanta
gtimurturkan3@gatech.edu

Joy Arulraj
Georgia Institute of Technology
Atlanta
arulraj@gatech.edu

ABSTRACT

We will demonstrate a prototype of SQLCHECK, a holistic toolchain for automatically finding and fixing anti-patterns in database applications. The advent of modern database-as-a-service platforms has made it easy for developers to quickly create scalable applications. However, it is still challenging for developers to design performant, maintainable, and accurate applications. This is because developers may unknowingly introduce anti-patterns in the application's SQL statements. These anti-patterns are design decisions that are intended to solve a problem, but often lead to other problems by violating fundamental design principles.

SQLCHECK leverages techniques for automatically: (1) detecting anti-patterns with high accuracy, (2) ranking them based on their impact on performance, maintainability, and accuracy of applications, and (3) suggesting alternative queries and changes to the database design to fix these anti-patterns. We will demonstrate that SQLCHECK enables developers to create more performant, maintainable, and accurate applications. We will show the prevalence of these anti-patterns in a large collection of queries and databases collected from open-source repositories.

PVLDB Reference Format:
Arthita Ghosh, Deven Bansod, Arpit Narechania, Prashanth Dintyala, Su Timurturkan, and Joy Arulraj. Interactive Demonstration of SQLCheck. PVLDB, 14(12): 2779-2782, 2021.
doi:10.14778/3476311.3476343

1 INTRODUCTION

Modern database applications produce qualitatively better insights in many domains, such as science, governance, and industry [4]. Two trends have simplified the design and deployment of such data-intensive applications. The first trend is the spread of data science skills to a larger community of developers [9, 18]. The second trend is the proliferation of database-as-a-service (DBaaS) platforms in the cloud [3, 12].

CHALLENGE: Designing applications is, however, non-trivial since applications may suffer from *anti-patterns* [11]. An anti-pattern (AP) is a design decision that is intended to solve a problem, but that often leads to other problems. `APs` lead to convoluted logical and physical database designs, thereby affecting the performance, maintainability, and accuracy of the application. The spread of data science skills to a larger community of developers places increased demand for a toolchain that facilitates application design without `APs`. Furthermore, the proliferation of DBaaS platforms obviates the need for in-house DBAs who used to assist application developers with finding and fixing these `APs`.

OUR APPROACH: To address this challenge, in our prior work [5], we presented a toolchain, called SQLCHECK, that assists application developers by: (1) detecting `APs` with high accuracy, (2) ranking the detected `APs` based on their impact, and (3) suggesting fixes for high-impact `APs`. The main thrust of our approach is to augment code analysis with data analysis (*i.e.*, examine both queries and data sets of the application) to detect `APs` with high precision and recall. We study the impact of frequently occurring `APs` on the key metrics of the application. We then use this information to rank the `APs` based on their estimated impact. By targeting frequently occurring `APs`, we take advantage of our ranking model trained on data collected from previous deployments without needing to share sensitive data (*e.g.*, data sets). Lastly, SQLCHECK suggests fixes for high-impact `APs` using rule-based query refactoring techniques.

This demonstration will showcase how SQLCHECK suggests fixes for high-impact `APs` using rule-based query refactoring techniques. Our demo will illustrate that SQLCHECK enables developers to create more performant, maintainable, and accurate applications. We will also show the prevalence of these anti-patterns in a large collection of queries and databases collected from open-source repositories. Users will be able to interact with SQLCHECK by submitting new queries and analysing the detected `APs`. The demonstration of SQLCHECK is available at [6].

2 DEMO SYSTEM

2.1 Workflow

Figure 1 illustrates the architecture of SQLCHECK. We envision that an user will use SQLCHECK in the following manner. A developer will deploy SQLCHECK on their local machine and connect it to the target application (*i.e.*, queries and database). ❶ The first component of

*These authors contributed equally to this work.
†This work was done when author(s) were at Georgia Institute Of Technology

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476343

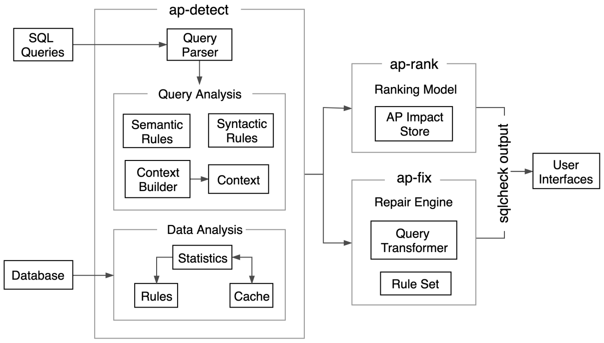


Figure 1: Architecture of sqlcheck: It takes in a SQL query and a connection to a DBMS (optional), and produces a ranked list of APs and associated fixes. Internally, sqlcheck leverages query and data analysis to detect the APs. It then uses a ranking model and a query repair engine to generate the desired fixes.

sqlcheck, ap-detect, performs static analysis of the queries to detect APs. To increase precision and recall, ap-detect also profiles the application's data and meta-data. ❷ Next, ap-rank examines the APs detected by ap-detect in the target application and ranks them based on their estimated impact. ❸ The third tool, ap-fix, suggests fixes for the high-impact APs identified by ap-rank using rule-based query transformations. ❹ Lastly, sqlcheck optionally uploads the APs detected in the application to an online AP repository with the permission of the developer. As new performance data is collected over time, we will retrain the ranking model of ap-rank to improve the quality of its decisions.

2.2 Interfaces

Our demo is implemented in Python [14] and exports three interfaces: (1) Interactive Shell, (2) REST, and (3) GUI. These interfaces are shown in Figure 2. Application developers and SQL IDE developers may leverage these interfaces to either directly interact with sqlcheck or to integrate it with their own IDEs. We describe these interfaces below:

- **Interactive Shell:** An SQL application developer can import the sqlcheck package from a package repository (*e.g.*, PyPI [15]) and directly use the interactive shell interface to execute SQL queries or leverage these sub-modules in other tools.

```
# Import the SQLCheck module
from sqlcheck.finder import find_anti_patterns
query = `INSERT INTO Users VALUES (1, 'foo')`
results = find_anti_patterns(query)
```

- **REST Interface:** This interface allows developers to leverage sqlcheck in applications developed in other programming languages by using web requests via HTTP. We implement this using the Flask web framework [13].

```
HTTP POST /api/check
Body: {"query":"INSERT INTO Users VALUES (1,'foo')"}

# Output of SQLCheck
{
  "query_analysis": [
    {
      "query": "INSERT INTO users VALUES (1,'foo','bar',25)
     ",
      "query_type": "DML",
      "anti_patterns": [
        {
          "impact": 0.9, # Performance Impact
          "fix": "Specify column names to avoid mismatch
     during insertion.
          For eg. 'INSERT INTO users (uid, fname, lname,
     age) VALUES (1,'foo','bar',25)'",
          "name": "Implicit Column Names"
        }
      ]
    }
  ],
  "data_analysis": {}
}
```

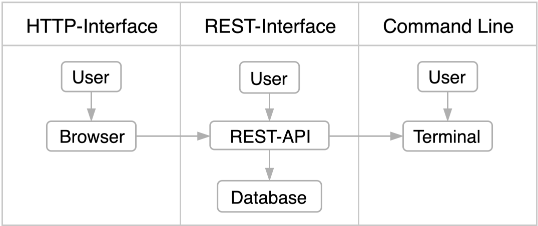


Figure 2: sqlcheck Interfaces – sqlcheck exports three interfaces: (1) command-line, (2) REST, and (3) HTTP.

- **GUI Interface:** Lastly, this interface is geared towards a wider range of users who are not familiar with application programming. This interface enables users to easily get feedback on their queries by copying them into the input field and is developed using ReactJS [8]. Internally, this invokes the REST interface that subsequently passes the queries to the sqlcheck binary which processes them and returns the list of detected APs and their associated fixes. This response is presented to the user through a ReactJS GUI [8].

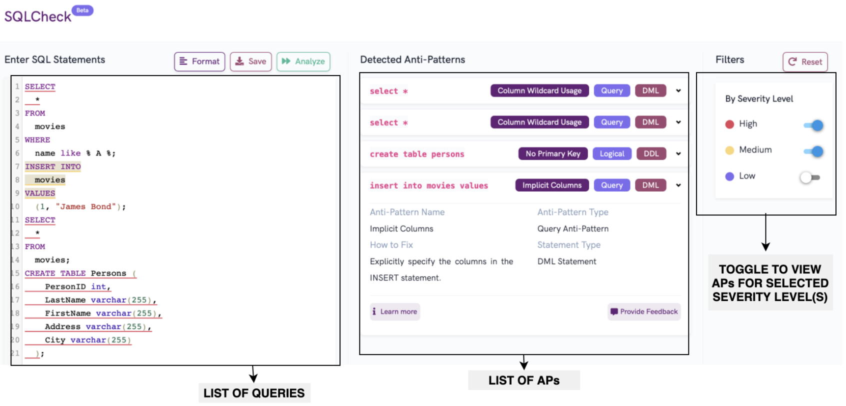
Extensibility: sqlcheck is extensible by design. A developer may add a new AP rule that implements the generic rule interface (name, type, detection rule, ranking metrics, and repair rule) and register it in the sqlcheck rule registry. A developer may also extend the context builder to augment the application's context for supporting complex rules. Lastly, a developer may replace the non-validating parser with a DBMS-specific parser to increase the utility of the parse tree.

2.3 Types of Anti-Patterns

We compiled a catalog of APs based on several resources that discuss best practices for schema design and querying DBMSs [7, 10, 11, 19]. Table 1 lists a subset of APs that sqlcheck targets. These APs fall under four categories:

❶ **Logical Design APs:** This category of APs arises from violating logical design principles that suggest the best way to organize data and the relationships that exist between them [17].

The *adjacency list* AP falls under this category. It refers to references between two attributes within the same table. Such a logical design is used to model hierarchical structures (*e.g.*, employee-manager relationship). With this representation, however, it is not



```
SELECT
  *
FROM
  movies
WHERE
  name like % A %;
INSERT INTO
  movies
VALUES
  (1, "James Bond");
SELECT
  *
FROM
  movies;
CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
  );
```

select *	Column Wildcard Usage	Query	DML
select *	Column Wildcard Usage	Query	DML
create table persons	No Primary Key	Logical	DDL
insert into movies values	Implicit Columns	Query	DML

Anti-Pattern Name: Implicit Columns
Anti-Pattern Type: Query Anti-Pattern
How to Fix: Explicitly specify the columns in the INSERT statement.
Statement Type: DML Statement

Learn more
Provide Feedback

By Severity Level
● High
● Medium
● Low

Figure 3: GUI Interface – Interface exported to the user.

Category	Anti-Pattern Name	Description	P	M	DA	DI	A
Logical Design APs	Multi-Valued Attribute	Storing list of values in a delimiter-separated list violating 1-NF.	✓	✓	✓$^{(\downarrow)}$	✓	✓
	No Primary Key	Lack of data integrity constraints.	✓	✓	✓$^{(\uparrow)}$	✓	-
	No Foreign Key	Lack of referential integrity constraints.	✓	✓	-	✓	-
	Generic Primary Key	Creating a generic primary key column (*e.g.*, `id`) for each table.	-	✓	-	-	-
	Adjacency List	Foreign key constraint referring to an attribute in the same table.	✓	-	-	-	-
Physical Design APs	Rounding Errors	Storing fractional data using a type with finite precision (*e.g.*, `FLOAT`).	-	-	-	-	✓
	Enumerated Types	Using `enum` to constrain the domain of a column.	✓	✓	✓$^{(\downarrow)}$	-	-
	External Data Storage	Storing file paths instead of actual file content in database.	-	✓	-	✓	✓
	Clone Table	Multiple tables matching the pattern <TableName>_N	✓	✓	-	✓	✓
Query APs	Column Wildcard Usage	Selecting all attributes from a table using wildcards to reduce typing.	✓	-	-	-	✓
	Concatenate Nulls	Concatenating columns that might contain `NULL` values using `||`.	-	-	-	-	✓
	Ordering by `RAND`	Using `RAND` function for random sampling or shuffling.	✓	-	-	-	-
	Pattern Matching	Using regular expressions for pattern matching complex strings.	✓	-	-	-	-
	Implicit Columns	Not explicitly specifying column names in data modification operations.	-	✓	-	✓	-
Data APs	Missing Timezone	Date-time fields stored without timezone.	-	-	-	-	✓
	Incorrect Data Type	Actual data does not conform to expected data type.	✓	-	✓$^{(\downarrow)}$	-	-
	Denormalized Table	Duplication of values.	✓	-	✓$^{(\downarrow)}$	-	-
	Information Duplication	Derived columns (*e.g.*, age from date of birth).	-	✓	-	✓	✓

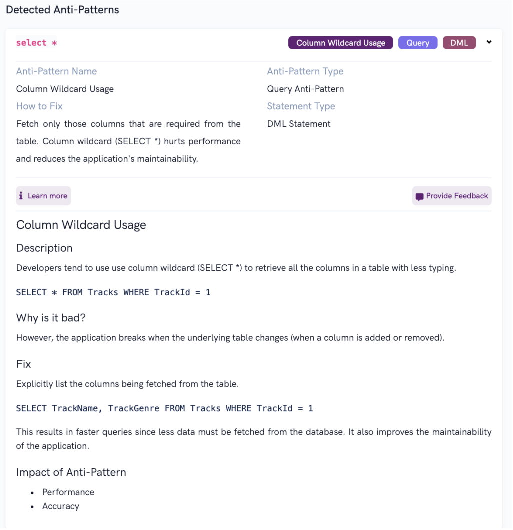
Table 1: List of Anti-Patterns: A catalog of `APs` based on best practices for database application design [7, 10, 11, 19]. They fall under four categories: (1) logical design `APs`, (2) physical design `APs`, (3) query `APs`, and (4) data `APs`. For each `AP` we illustrate its impact on five metrics: (1) Performance (P), (2) Maintainability (M), (3) Data Amplification (DA), (4) Data Integrity (DI), and (5) Accuracy (A). ✓ represents that the given `AP` affects that metric. ↑ and ↓ refer to increase and decrease in data amplification, respectively, when that `AP` is fixed.

trivial to handle common tasks such as retrieving the employees of a manager up to a certain depth and maintaining the integrity of the relationships when a manager is removed.

❷ **Physical Design `APs`:** The next category of `APs` is associated with efficiently implementing the logical design using the features of a DBMS. This includes *rounding errors* and *enumerated types* `APs`. The rounding errors `AP` arises when a scientist uses a type with finite precision, such as `FLOAT` to store fractional data. This may introduce accuracy problems in queries that calculate aggregates. The enumerated types `AP` occurs when a scientist restricts a column's values by specifying the fixed set of values it can take while defining the table. However, this `AP` makes it challenging to add, remove, or modify permitted values later and reduces the application's portability.

❸ **Query `APs`:** Query `APs` arise from violating practices that suggest the best way to retrieve and manipulate data using SQL. This includes *`NULL` usage* and *column wildcard usage* `APs`. Developers are often caught off-guard by the behavior of `NULL` in SQL. Unlike in most programming languages, SQL treats `NULL` as a special value, different from zero, false, or an empty string. This results in counter-intuitive query results and introduces accuracy problems. The latter `AP` arises when a developer uses wildcards (`SELECT *`) to retrieve all the columns in a table with less typing. This `AP`, however breaks applications on refactoring.

❹ **Data `APs`:** Data `APs` are a subset of `APs` that sqlcheck detects by analysing the data (as opposed to queries). This includes the *incorrect data type* and *information duplication* `APs`. The former `AP` arises due to data type mismatches (*e.g.*,, storing a numerical field in a TEXT column). This negatively impacts performance and leads to data amplification. The latter `AP` occurs when a column contains data derived from another column in the same table (*e.g.*, storing `age` based on `date of birth`). While this accelerates query processing, it reduces maintainability and leads to data amplification.



Detected Anti-Patterns

```
select *
```

Anti-Pattern Name
Column Wildcard Usage

Anti-Pattern Type
Query Anti-Pattern

How to Fix
Fetch only those columns that are required from the table. Column wildcard (SELECT *) hurts performance and reduces the application's maintainability.

Statement Type
DML Statement

Column Wildcard Usage

Description

Developers tend to use use column wildcard (SELECT *) to retrieve all the columns in a table with less typing.

```
SELECT * FROM Tracks WHERE TrackId = 1
```

Why is it bad?

However, the application breaks when the underlying table changes (when a column is added or removed).

Fix

Explicitly list the columns being fetched from the table.

```
SELECT TrackName, TrackGenre FROM Tracks WHERE TrackId = 1
```

This results in faster queries since less data must be fetched from the database. It also improves the maintainability of the application.

Impact of Anti-Pattern

- Performance
- Accuracy

Figure 4: Fix Card – suggested fix for column wildcard usage AP.

3 DEMO SCENARIOS

The demonstration of sqlcheck is available at [6]. The goals of this demonstration are listed below:

- **Detecting APs**: As shown in Figure 3, we will illustrate how sqlcheck detects different types of APs listed in §2.3 using query and data analysis. sqlcheck also explains why a particular query suffers from an AP.
- **Ranking APs**: Given a set of APs, sqlcheck automatically ranks them based on their impact on key metrics of the database application and classifies them into three categories based on their impact. The user may view a subset of the detected APs by configuring the severity level in the demo. This allows them to prioritize their attention on high-impact APs.
- **Fixing APs**: Lastly, we will illustrate the fixes suggested by sqlcheck. For instance, the fix for the *column wildcard usage* AP is shown in Figure 4.

4 RELATED WORK

Transforming Database Applications: While program analysis has been widely used in software engineering, it has not been extensively utilized by the DBMS community. Recent efforts have focused on transforming database-backed programs to improve performance [2, 21]. DBridge presents a set of holistic optimizations including query batching and binding, and automatic transformation of object-oriented code into synthesized queries [16]. Cheung *et al.* describe techniques for batching queries to reduce the number of round trips between the application and the database server [1].

Object-Relational Mapping: Researchers have studied the impact of ORM on application design and performance [20]. Yang *et al.* perform a comprehensive study of performance issues in database applications using profiling [22]. sqlcheck is the first effort focused on exploring the problems of automatically ranking and fixing APs in database applications.

5 CONCLUSION

We demonstrate sqlcheck, a holistic toolchain for finding, ranking, and fixing APs in database applications. sqlcheck leverages a novel AP detection algorithm that augments query analysis with data analysis. It uses the overall context of the application to reduce false positives and negatives. sqlcheck relies on a ranking model for characterizing the impact of detected APs and suggests fixes for high-impact AP using rule-based query refactoring techniques. Our empirical analysis shows that sqlcheck enables developers to create more performant, maintainable, and accurate applications.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. National Science Foundation (IIS-1908984, IIS-1850342), Alibaba, Cisco, Intel, and Adobe. We thank Shamkant Navathe, Karthik Ramachandra, and the reviewers for their constructive feedback. We thank all of the contributors to sqlcheck.

REFERENCES

[1] Alvin Cheung, Owen Arden, Samuel Madden, Armando Solar-Lezama, and Andrew C. Myers. 2013. StatusQuo: Making Familiar Abstractions Perform Using Program Analysis. In *Proc. of CIDR*.
[2] Alvin Cheung, Samuel Madden, Armando Solar-Lezama, Owen Arden, and Andrew C Myers. 2014. Using Program Analysis to Improve Database Applications. *IEEE Data Eng. Bull.* 37, 1 (2014), 48–59.
[3] Carlo Curino, Evan PC Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Sam Madden, Hari Balakrishnan, and Nickolai Zeldovich. 2011. Relational cloud: A database-as-a-service for the cloud. (2011).
[4] Thomas H Davenport and DJ Patil. 2012. Data scientist. *Harvard business review* 90, 5 (2012), 70–76.
[5] Prashanth Dintyala, Arpit Narechania, and Joy Arulraj. 2020. SQLCheck: Automated Detection and Diagnosis of SQL Anti-Patterns. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2331–2345.
[6] Georgia Tech Database Group. 2021. SQLCheck Demo. http://db-apps.cc.gatech.edu/sqlcheck/playground.
[7] Cunningham & Cunningham Inc. 2014. C2 Wiki. http://wiki.c2.com/?AntiPatternsCatalog.
[8] Facebook Inc. 2019. ReactJS. https://reactjs.org.
[9] Stitch Inc. 2018. The State of Data Science. https://www.stitchdata.com/resources/the-state-of-data-science/.
[10] Stack Exchange Inc. 2010. StackOverflow Wiki. https://stackoverflow.com/questions/346659/what-are-the-most-common-sql-anti-patterns.
[11] Bill Karwin. 2010. *SQL antipatterns: avoiding the pitfalls of database programming.* Pragmatic Bookshelf.
[12] David Lomet, Alan Fekete, Gerhard Weikum, and Mike Zwilling. 2009. Unbundling transaction services in the cloud. *arXiv preprint arXiv:0909.1768* (2009).
[13] Pallets. 2019 . Python-Flask. http://flask.palletsprojects.com/en/1.1.x.
[14] Python Software Foundation. 2019 . Python. https://www.python.org.
[15] Python Software Foundation. 2019. PyPi. https://pypi.org.
[16] Karthik Ramachandra, Mahendra Chavan, Ravindra Guravannavar, and S Sudarshan. 2015. Program transformations for asynchronous and batched query submission. *In TKDE* 27, 2 (2015), 531–544.
[17] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database Management Systems* (3 ed.). McGraw-Hill, Inc., New York, NY, USA.
[18] Toby Segaran and Jeff Hammerbacher. 2009. *Beautiful data: the stories behind elegant data solutions.* " O'Reilly Media, Inc.".
[19] Tushar Sharma, Marios Fragkoulis, Stamatia Rizou, Magiel Bruntink, and Diomidis Spinellis. 2018. Smelly relations: measuring and understanding database schema quality. In *Proc. of ICSE*. ACM, 55–64.
[20] Alexandre Torres, Renata Galante, Marcelo S Pimenta, and Alexandre Jonatan B Martins. 2017. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information and Software Technology* 82 (2017), 1–18.
[21] Cong Yan and Alvin Cheung. 2016. Leveraging lock contention to improve OLTP application performance. *In Proceedings of VLDB* 9, 5 (2016), 444–455.
[22] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. 2018. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *Proc. of ICSE*. IEEE, 800–810.