SQLCheck: Automated Detection and Diagnosis of SQL Anti-Patterns

Prashanth Dintyala*
vdintyala3@gatech.edu
Georgia Institute of Technology

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

Joy Arulraj
arulraj@gatech.edu
Georgia Institute of Technology

Abstract

The emergence of database-as-a-service platforms has made deploying database applications easier than before. Now, developers can quickly create scalable applications. However, designing performant, maintainable, and accurate applications is challenging. 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.

In this paper, we present SQLCHECK, a holistic toolchain for automatically finding and fixing anti-patterns in database applications. We introduce techniques for automatically (1) detecting anti-patterns with high precision and recall, (2) ranking the anti-patterns 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 demonstrate the prevalence of these anti-patterns in a large collection of queries and databases collected from open-source repositories. We introduce an anti-pattern detection algorithm that augments query analysis with data analysis. We present a ranking model for characterizing the impact of frequently occurring anti-patterns. We discuss how SQLCHECK suggests fixes for high-impact anti-patterns using rule-based query refactoring techniques. Our experiments demonstrate that SQLCHECK enables developers to create more performant, maintainable, and accurate applications.

CCS Concepts

• **Information systems** → **Database utilities and tools**.

Keywords

Anti-Patterns; Database Applications

---

*These authors contributed equally to this work.

, ,
.

1 Introduction

Two major trends have simplified the design and deployment of data-intensive applications. The first is the spread of data science skills to a larger community of developers [24, 39]. Data scientists combine rich data sources in applications that process large amounts of data in real-time. These applications produce qualitatively better insights in many domains, such as science, governance, and industry [14]. The second trend is the proliferation of database-as-a-service (DBaaS) platforms in the cloud [13, 30]. Due to economies of scale, these services enable greater access to database management systems (DBMSs) that were previously only in reach for large enterprises. DBaaS platforms obviate the need for in-house database administrators (DBAs) and enable data scientists to quickly deploy widely used applications.

CHALLENGE: Designing database applications is, however, non-trivial since applications may suffer from *anti-patterns* [27]. An anti-pattern (AP) refers to a design decision that is intended to solve a problem, but that often leads to other problems by violating fundamental design principles. APs in database applications can 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 since scientists who are experts in other domains are likely not familiar with these anti-patterns [24, 39]. Furthermore, the proliferation of DBaaS platforms obviates the need for in-house DBAs who used to assist application developers with finding and fixing APs.

Sharma *et al.* have designed a tool, called DBDEO, for automatically detecting APs in database applications [41]. They demonstrate the widespread prevalence of APs in production applications. Although their detection algorithm of DBDEO is effective in uncovering APs, it suffers from three limitations. First, the static analysis algorithm suffers from low precision and recall. Second, it does not rank the APs based on their impact. Third, it does not suggest solutions for fixing them. Thus, a developer would need to manually confirm the APs detected by DBDEO, identify the high-impact APs among them, and fix them.

In this paper, we investigate how to find, rank, and fix APs in database applications. We present a toolchain, called SQLCHECK, that assists application developers by: (1) detecting APs with high precision and recall, (2) ranking the detected APs based on their impact on performance and maintainability, (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 performance, maintainability, and accuracy 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. The advantage of our approach over DBDEO is that it reduces the time that a developer must expend on identifying high-impact APs and fixing them.

In summary, we make the following contributions:

- We illustrate the limitations of the state-of-the-art tools for identifying APs in database applications and motivate the need for an alternate approach with higher precision and recall (§2).

- We introduce an AP detection algorithm that augments query analysis with data analysis (§4).

- We present a ranking model for characterizing the impact of frequently occurring APs on the performance, maintainability, and accuracy of the application (§5).

- We discuss how SQLCHECK suggests fixes for high-impact APs using rule-based query refactoring techniques (§6).

- We illustrate the efficacy of SQLCHECK in finding, ranking, and fixing APs through an analysis of 1406 open-source database applications, 15 Django applications, 31 Kaggle databases, and a user study (§8).

2 Motivation & Background

We illustrate the need for detecting and diagnosing APs through a case study, then present an overview of the different types of APs and conclude with a discussion on the impact of APs on the application's performance, maintainability, and accuracy.

2.1 Case Study: GlobaLeaks

We illustrate the problems introduced by APs through a case study of GlobaLeaks, an open-source application for anonymous-whistleblowing initiatives [17]. The application

Tenant_ID	Zone_ID	Active	User_IDs
T1	Z1	True	U1 , U2
T2	Z3	True	U3 ; U4

(a) Tenants Table

User_ID	Name	Role	Email
U1	N1	R1	E1
U2	N2	R2	E2
U3	N3	R3	E3
U4	N4	R4	E4

(b) Users Table

Figure 1: GlobaLeaks Application – List of tables.

supports a *multi-tenancy* feature to enable multiple organizations to accept submissions and direct them to different endpoints within a single deployment of the application.

EXAMPLE 1: Figure 1 presents the logical database design of the two tables associated with this feature[1]. Since a given tenant can serve multiple users (*i.e.*, one-to-many relationship), the application developer decided to store this information as a comma-separated list of user identifiers in the `User_IDs` column of `Tenants` table. While this *multi-valued attribute* design pattern captures the relationship between the two entities without introducing additional tables or columns, it suffers from performance, maintainability, and data integrity problems. We illustrate these problems using a set of tasks and associated SQL queries executed by the application.

TASK #1: The developer is interested in retrieving the tenants that a user is associated with. We cannot use the equality operator in SQL to solve this task since the users are stored in a comma-separated list. Instead, we must employ *pattern-matching expressions* to search for that user:

```
/* List the tenants that a user is associated with */
SELECT * FROM Tenants WHERE User_IDs LIKE `[[:<:]]U1[[:>:]]';
```

TASK #2: Consider the task of retrieving information about the users served by a tenant. This query is also computationally expensive since this involves joining the comma-separated list of users to matching rows in the `Users` table. Joining two tables using an expression prevents the DBMS from using indexes to accelerate query processing [18]. Instead, it must scan through both tables, generate a cross product, and evaluate the regular expression for every combination of rows.

```
/* Retrieve users served by a tenant */
SELECT * FROM Tenants AS t JOIN Users AS u
ON t.User_IDs LIKE `[[:<:]]'||u.User_ID||`[[:\>:]]'
WHERE t.Tenant_ID = 'T1';
```

DATA INTEGRITY PROBLEMS: Another major limitation of this approach is that the developer implicitly assumes that

---

[1]We distilled the essence of this AP for the sake of presentation.

User_ID	Name	Role	Email
U1	N1	R1	E1

(a) Users Table

Tenant_ID	Zone_ID	Active
T1	Z1	True
T2	Z2	True

(b) Tenants Table

Tenant_ID	User_ID
T1	U1
T1	U2
T2	U3
T2	U4

(c) Hosting Table

Figure 2: Refactored GlobaLeaks Application – List of tables.

users will be stored as a list of strings separated by a comma. This implicit assumption might later be violated by a developer entering users separated by another delimiter, such as a semi-colon (*e.g.* "U6; U7"). This is feasible since the DBMS is not explicitly enforcing that the string should be separated by a particular character. Given this new data, the developer must update all the queries operating on the `User_IDs` column to handle the usage of multiple separator characters. Furthermore, it is not feasible for the DBMS to enforce a referential integrity constraint between these columns: (1) `User_IDs` in `Tenants`, and (2) `User_ID` in `Users`. This is because the former column encodes the comma-separated list as a string. So, it is possible for a user in the former column to not have a corresponding tuple in the latter column.

2.1.1 Solution: Intersection Table We can eliminate this AP by creating an additional *intersection table* to encode the many-to-many relationship between tenants and users [2]. This table references the `Tenant` and `User` tables. In Figure 2, the `Hosting` table implements this relationship between the two referenced tables.

```
/* Create an intersection table */
CREATE TABLE Hosting (
    User_ID VARCHAR(10) REFERENCES User(User_ID),
    Tenant_ID VARCHAR(10) REFERENCES Tenants(Tenant_ID),
    PRIMARY KEY (User_ID, Tenant_ID)
);
/* Drop redundant column */
ALTER TABLE Tenants DROP COLUMN User_IDs;
```

We will next illustrate how this AP-free logical design enables simpler queries for all of the tasks.

TASKS #1 AND #2: It is straightforward to join the `Tenants` and `Users` tables with the `Hosting` table to solve the first two tasks. These queries are easy to write for developers and easy to optimize for DBMSs. The DBMS can now use an index on `User_IDs` to efficiently execute the join instead of matching regular expressions.

```
/* List the tenants that a user is associated with */
SELECT * FROM Hosting as H JOIN Tenants as T
ON H.Tenant_ID == T.Tenant_ID WHERE H.User_ID = 'U1';
```

[2] That is, each user may be associated with multiple tenants, and likewise each tenant may serve multiple users.

```
/* Retrieve information about users served by tenant */
SELECT * FROM Hosting as H JOIN Tenants as T
ON H.User_ID == T.User_ID WHERE H.Tenant_ID = 'T1';
```

DATA INTEGRITY PROBLEMS: The developer can delegate the task of ensuring data integrity to the DBMS by specifying the appropriate foreign key constraints. The DBMS will enforce these constraints when data is ingested or updated.

2.2 Classification of Anti-Patterns

We compiled a catalog of APs based on several resources that discuss best practices for schema design and query structuring [21, 25, 27, 41]. Table 1 lists the 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 and interconnect data. It includes the *multi-valued attribute* AP covered in §2.1. The *adjacency list* AP also 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 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 [3].

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

[3] `ENUM` data type is a proprietary feature in the MySQL DBMS.

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.	✓	✓	✓(↓)	✓	✓
	No Primary Key	Lack of data integrity constraints.	✓	✓	✓(↑)	✓	-
	No Foreign Key	Lack of referential integrity constraints.	✓	✓	-	✓	-
	Generic Primary Key	Creating a generic primary key column (e.g., id) for each table.	-	✓	-	-	-
	Data In Metadata	Hard-coding application logic in table's meta-data.	✓	✓	✓(↓)	✓	✓
	Adjacency List	Foreign key constraint referring to an attribute in the same table.	✓	-	-	-	-
	God Table	Number of attributes defined in the table cross a threshold (e.g., 10)	✓	✓	-	-	-
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.	✓	✓	✓(↓)	-	-
	External Data Storage	Storing file paths instead of actual file content in database.	-	✓	-	✓	✓
	Index Overuse	Creating too many infrequently-used indexes.	✓	✓	✓(↓)	-	-
	Index Underuse	Lack of performance-critical indexes.	✓	✓	✓(↑)	-	-
	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.	-	✓	-	✓	-
	DISTINCT and JOIN	Using DISTINCT to remove duplicate values generated by a JOIN.	✓	✓	-	-	-
	Too Many Joins	Number of JOINs cross a threshold.	✓	-	-	-	-
Data APs	Missing Timezone	Date-time fields stored without timezone.	-	-	-	-	✓
	Incorrect Data Type	Actual data does not conform to expected data type.	✓	-	✓(↓)	-	-
	Denormalized Table	Duplication of values.	✓	-	✓(↓)	-	-
	Information Duplication	Derived columns (e.g., age from date of birth).	-	✓	-	✓	✓
	Redundant Column	Column with NULLS or same value (e.g., en-us)	-	-	✓(↓)	-	-
	No Domain Constraint	All values should belong to particular range (e.g., rating)	-	✓	✓(↓)	✓	-

Table 1: List of Anti-Patterns: A catalog of APs based on best practices for database application design [21, 25, 27, 41]. 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.

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.

2.3 Impact of Anti-Patterns

APs in database applications lead to convoluted logical and physical database designs, thereby affecting the performance, maintainability, and accuracy of the application.

1. PERFORMANCE: An application's performance is often measured in terms of throughput (*e.g.* the number of requests that can be processed per second) and latency (*e.g.* the time that it takes for the system to respond to a request) [38]. Optimizing these metrics is important because they determine how quickly an application can process data and how quickly a user can leverage the application to make new decisions.

Consider the tasks presented in §2.3. We measured the impact of the *multi-valued attribute* APs on the time taken to
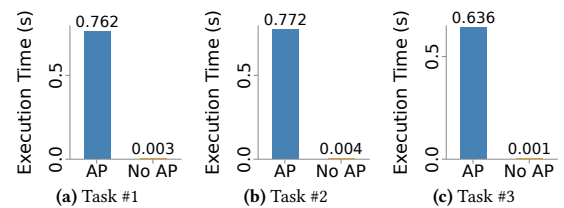

(a) Task #1 **(b)** Task #2 **(c)** Task #3

Figure 3: Multi-Valued Attribute AP: Performance impact of the multi-valued attribute AP on the above-mentioned tasks.

execute these tasks. Figure 3 presents the results of this experiment. We defer the discussion of the experimental setup to §8.2. When we remove this AP, the queries associated with these tasks accelerated by 636×, 256×, and 193× respectively. These results illustrate the importance of fixing APs.

2. MAINTAINABILITY: The maintainability of an application represents the ease with which the application's design and component queries can be modified to adapt to a changed environment, improve performance or other metrics, or correct faults [20]. Maintainable applications allow developers to quickly and easily add new features, fix bugs in existing features, and increase performance.
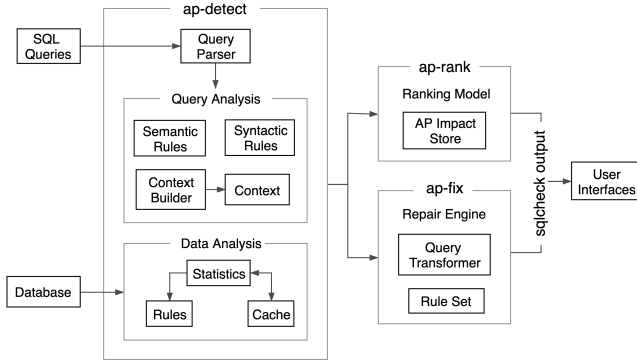
Figure 4: Architecture of sqlcheck: It takes in a SQL query and database (optional), and produces a ranked list of APs and associated fixes. Internally, sqlcheck leverages query and data analysis to detect the APs. It then uses a ranking model and query repair engine to generate the desired fixes.

3. Accuracy: An application's accuracy is measured in terms of the discrepancy between the data stored by a user and that returned by the application. For example, an application that stores fractional numeric data using the FLOAT type in SQL can fail to return certain tuples due to slight discrepancies in their values [27].

Given the impact of APs, we next present an overview of a toolchain that assists developers in eliminating them.

3 System Overview

sqlcheck is geared towards automatically finding, ranking, and fixing APs in database applications. Application developers can leverage sqlcheck to create more performant, maintainable, and accurate database applications. sqlcheck contains three components for: (1) detecting APs with high precision and recall, (2) ranking the detected APs based on their impact, and (3) suggesting fixes to application developers for high-impact APs.

Workflow: We envision that application developers 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 data sets). ❶ The first component, ap-detect, will then perform static analysis of the queries to detect APs. To increase precision and recall, ap-detect will profile the application's data and meta-data (§4). ❷ Next, ap-rank will examine the APs detected by ap-detect in the target application and rank them based on their estimated impact (§5). ❸ The third tool, ap-fix, will suggest fixes for the high-impact APs identified by ap-rank using rule-based query transformations (§6). ❹ Lastly, sqlcheck will optionally upload 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, ap-rank will retrain its ranking model to improve the quality of its decisions.

Algorithm 1: sqlcheck Algorithm

input : application queries Q, database $\mathcal{D}$
output: detected APs and associated fixes
  // Extract context from queries
1   query context $QC \leftarrow \{ \}$
2   **for** *query q in Q* **do**
3   QC.append(Query-Analyser(q))
  // Extract context from data
4   data context $\mathcal{D}C \leftarrow \{ \}$
5   **for** *table t in $\mathcal{D}$.tables* **do**
6      $\mathcal{D}C$.append(Data-Analyser(t))
7   context C = Context-Builder(QC, $\mathcal{D}C$)
  // Detect anti-patterns
8   anti-patterns $\mathcal{P} \leftarrow \{ \}$
9   **for** *query q in Q* **do**
10     $\mathcal{P}$.append(Query-Rules(q, C))
11   **for** *table t in $\mathcal{D}$.tables* **do**
12     $\mathcal{P}$.append(Data-Rules(t, C))
13   ranked anti-patterns $\mathcal{R} \leftarrow$ AP-Rank($\mathcal{P}$)
14   **return** AP-Fix($\mathcal{R}$)

---

4 Finding Anti-Patterns

In this section, we present the techniques used by ap-detect for identifying APs in a given application. We begin with an overview of the sqlcheck algorithm.

Overview: As shown in Algorithm 1, ap-detect takes a list of SQL queries executed by the application and a connection to the database server as input. It constructs an *application context* using the given inputs. First, it uses a *query analyser* to extract context from the queries (*e.g.*, column names, table names, predicates, constraints, and indexes). It tailors the analysis based on the type of SQL statement. We present the query analyser in §4.1. Next, it uses a *data analyser* to extract context from the tables in the database (*e.g.*, data distribution and format of each column). We present the data analyser in §4.2. Based on the constructed application context, ap-detect uses a set of rules for identifying APs in the given queries[4]. These rules are general-purpose functions that leverage the overall context of the application: (1) queries, (2) data, and (3) meta-data. The APs identified using such query and data analyses are then ordered by ap-rank, which will be covered in §5.

4.1 Query Analysis

ap-detect begins by analysing the SQL queries Q executed by the target database application. It detects APs in the given queries in two phases: (1) intra-query detection and (2) inter-query detection. During the first phase, it identifies APs in each query q in Q. During the second phase, it leverages the entire context of the application (*i.e.*, other queries in Q

---

[4]This includes both data definition language (DDL) and data manipulation language (DML) commands [11].

and logical design of the database *D*) to detect more complex APs. We next discuss these techniques in detail.

❶ **Intra-Query Detection:** `ap-detect` applies a set of *query rules* on the given query *q*. Each query rule consists of a general-purpose function to identify the existence of the target AP in *q*. Rules can range from a simple, pattern-matching function that uses a set of regular expressions to complex functions that leverage the inferred context of the application. In order to support diverse SQL dialects, `ap-detect` leverages a non-validating SQL parser, called `sqlparse` [2], to process the SQL statement. This parser supports multiple dialects by virtue of its non-validating parsing logic. However, unlike a typical DBMS parser [33], it does not generate a semantically-rich parse tree. We address this limitation by annotating the parse tree returned by `sqlparse`. `ap-detect` and `ap-fix` use this annotated parse tree for finding APs and suggesting fixes, respectively. The tree-structured representation (as opposed to the raw SQL string) allows recursive application of rules and improves the extensibility of the rule system.

Example 2: Consider the following SQL statement for inserting a record into the `TENANT` table:

```sql
INSERT INTO Tenant VALUES ('T1', 'Z1', True, 'U1,U2');
```

This DML statement fails to correctly function when the schema of `TENANT` table evolves. For instance, if we drop the `User_IDs` column and add a new column termed `Description`, it would incorrectly insert values into the table. This *implicit column usage* AP also reduces the maintainability of the application. This is because explicitly specifying the column names improves the readability of the query for another developer who is trying to infer the values being inserted into each column. `ap-detect` identifies this AP by checking whether the column names are present in the `INSERT` statement. An intra-query detection rule is sufficient to detect the existence of this AP. However, to suggest a fixed `INSERT` statement, `ap-fix` needs the application's context (*i.e.*, the schema of the `TENANT` table).

❷ **Inter-query detection:** The intra-query detection technique suffers from low precision and recall as it does not leverage the relationship between queries in the application which is critical for detecting complex APs.

Example 3: Consider the *No Foreign Key* AP in GlobaLeaks [17]. There are two tables: `TENANT` and `QUESTIONNAIRE`. The `Tenant_ID` column should connect these two tables. However, the DDL statement of the `QUESTIONNAIRE` table does not define this foreign key relationship. Since the intra-query detection technique applies the rules to each query independently, it is unable to detect this AP by separately examining the two DDL statements. `ap-detect` can detect the missing

---

Algorithm 2: Detecting Anti-Patterns via Query Analysis

---
input : application query *q*, context *C*
output: detected APs

1 anti-patterns $\mathcal{P} \leftarrow \{ \ \}$
   // anti-pattern detection rules based on type of query
2 rules $\mathcal{R} \leftarrow$ RulesForQuery(*q*)
3 **for** *rule r* **in** $\mathcal{R}$ **do**
4   anti-pattern $p \leftarrow r(q, C)$
       // use context to identify relevant contextual rules
5   contextual rules $\mathcal{F} \leftarrow$ RelevantRules(*p, q, C*)
       // use contextual rules to reduce false positives and negatives
6   **if** $\mathcal{F}(C, q, p)$ **then**
7         $\mathcal{P}$.append(*p*)
8 **return** $\mathcal{P}$

---

foreign key only if it considers both DDL statements along with the `JOIN` condition in the `SELECT` query as follows.

```sql
/* Tenant table */
CREATE TABLE Tenant(Tenant_ID INTEGER PRIMARY KEY,
Zone_ID VARCHAR(30) NOT NULL, Active BOOLEAN);
/* Questionnaire table */
CREATE TABLE Questionnaire (Questionnaire_ID UUID PRIMARY KEY,
Tenant_ID INTEGER, Name VARCHAR(30), Editable BOOLEAN);
/* Select query */
SELECT q.Name, q.Editable, t.Active
FROM   Questionnaire q JOIN Tenant T
ON T.Tenant_ID = Q.Tenant_ID WHERE q.Editable = true;
```

We address the limitations of the intra-query detection technique by constructing the application's *context*. The context contains two components: (1) the schema and (2) the queries associated with the application. The AP detection rules utilize the context to resolve cases where the presence or absence of an AP cannot be determined with high precision by only looking at a given query.

Algorithm 2 presents the algorithm used for detecting APs by analyzing queries. The `ContextBuilder` constructs the context using the analysed queries and the database. The context exports a queryable interface for applying contextual rules on the queries, schema, and other application-specific metadata. If the database is not available, the `ContextBuilder` leverages the DDL statements to construct the context. Given the context, `ap-detect` first applies a set of AP detection rules based on the type of the query. It then uses the context to identify the relevant context specific rules that are subsequently applied to reduce false positives and negatives.

Limitation: The inter-query detection technique also suffers from false positives and negatives. Consider the multi-valued attribute AP discussed in Example 1. `ap-detect` uses a pattern-matching rule (*i.e.*, regular expression) for detecting this AP in SELECT queries containing string processing tricks. However, this rule can result in: (1) false negatives if the delimiter-separated strings are handled externally in

Algorithm 3: Detecting Anti-Patterns via Data Analysis

---
    input   : context C, database $\mathcal{D}$
    output: detected APs
1 anti-patterns $\mathcal{P} \leftarrow \{\ \}$
2 **for** *rule d* **in** *data rules* $\mathcal{D}$ **do**
3   **for** *table t* **in** $\mathcal{D}$*.tables* **do**
         // sample tuples from the table
4   sampled tuples $s = \textsc{Sample}(t)$
         // use data rules to reduce false positives and negatives
5   **if** *r(C[t], s)* **then**
6           $\mathcal{P}$.append(p)
7 **return** $\mathcal{P}$

---

the application code, and (2) false positives if the delimiter is used for an alternate purpose in application (*e.g.*, `ADDRESS` attribute). Thus, it is not feasible to identify this `APs` with high precision and recall by only examining the SQL queries. We next discuss how `ap-detect` extracts and utilizes context from the tables in the database to overcome this limitation.

4.2 Data Analysis

`ap-detect` leverages data analysis to improve the precision and recall of anti-pattern detection. It uses a *data analyzer* to profile the contents of the database used by the application and uses it to augment the context described in §4.1. This information is also used for retrieving the relevant contextual rules while detecting `APs` via query analysis.

For example, in case of the multi-valued attribute `AP` (Example 1), `ap-detect` uses a *data rule* that checks whether a particular column contains delimiter-separated strings. It first checks the data type of the column. If the column is a `VARCHAR` or `TEXT` field, it samples the columnar data and checks whether that contains delimiter-separated strings. In case of the `TENANT` table, as shown in Figure 1a, the `User_-IDs` column contains comma-separated strings. Even if the query rules are unable to detect this `AP`, the data rule will correctly flag this column as suffering from the MVA `AP`.

The data analyzer first scans the database to collect: (1) the schemata of the component tables, and (2) the distribution of the data in the component columns (*e.g.*, unique values, mean, median, etc.). It then collects samples from each table in the examined database. `ap-detect` applies a set of rules for determining the existence of `APs` in the sampled data. If one of these rules is activated, then `ap-detect` appends the associated anti-pattern to the list of `APs` sent to `ap-rank`.

EXAMPLE 4: Consider the following `AP` in GlobaLeaks. The `Role` column in the `USER` table represents the roles assumed by the users. The developer chose to encode this data as a `STRING` field with a constraint on the field's domain.

```
ALTER TABLE User ADD CONSTRAINT User_Role_Check
CHECK (ROLE IN ('R1', 'R2', 'R3'));
```

We refer to this as the Enumerated Types `AP`. In this case, the data analyzer extracts the type information of the `Role` column and notices that it as a `STRING` field. It then samples the data in the column. `ap-detect` uses the context to compute the ratio of distinct values to the number of tuples. If this ratio is greater than a given threshold, it detects this `AP`.

Since data analysis is computationally expensive (*e.g.*, sampling), `ap-detect` reuses the constructed context across several checks. The data analyzer periodically refreshes the context over time. It also refreshes the context whenever the schema evolves. `ap-detect` allows the developer to configure the tuple sampling frequency and the thresholds associated with activating data rules.

RULE COMPLEXITY: `ap-detect` supports complex, general-purpose rules that leverage the overall context of the application. Example 5 illustrates the complexity of rules.

EXAMPLE 5: The *Index Overuse* `AP` is associated with the creation of too many infrequently-used indexes. For instance, consider these three indexes in the `TENANT` table.

```
CREATE INDEX idx_zone_actv (Zone_ID, Active); /* Index 1 */
CREATE INDEX idx_zone (Zone_ID); /* Index 2 */
CREATE INDEX idx_actv (Active); /* Index 3 */
/* Queries (Workload 1) */
SELECT Tenant_ID FROM Tenant WHERE Zone_ID = 'Z1'
 AND Active = 'True';
SELECT Tenant_ID FROM Tenant WHERE Tenant_ID = 'T1'
AND Active = 'True';
/* Queries (Workload 2) */
SELECT Tenant_ID FROM Tenant WHERE Zone_ID = 'Z1';
SELECT Tenant_ID FROM Tenant WHERE Active = 'True';
```

Depending on the workload, `ap-detect` marks different set of indexes as potentially exhibiting this `AP`. It leverages the context to determine the list of constructed indexes. For the first workload, it marks the second and third indexes as redundant since these queries will leverage the index on `Tenant_ID`. For the second workload, it marks the first index as redundant since these queries will leverage the second and third indexes. Thus, `ap-detect` supports complex rules.

5 Ranking Anti-Patterns

In this section, we present the algorithm used by `ap-rank` for ordering the `APs` identified by `ap-detect`. We begin with an overview of the metrics collected by `ap-rank` for ordering the `APs`. We then present the model used by `ap-rank`.

5.1 Metrics for Ranking Anti-Patterns

`ap-rank` collects six metrics for each `AP`. These metrics are subsequently used by the model for ordering the `APs`.

❶ **Read and Write Performance (RP, WP):** This metric characterizes the impact of the `AP` on the application's performance. We measure the time taken to execute different

Role_ID	Role_Name
1	R1
2	R2
3	R3

(a) Role Table

User_ID	Name	Role	Email
U1	N1	1	E1
U2	N2	2	E2
U3	N3	2	E3
U4	N4	3	E4

(b) User Table

Figure 5: Refactored GlobaLeaks Application – List of tables.

$$S_{rp}(x), S_{wp}(x), S_m(x) = \min(1, x/5)$$
$$S_{da}(x) = \min(1, x/8)$$
$$S_{di}(x), S_a(x) = x \; // \; x \in \{0, 1\}$$

$$\text{score} = W_{rp} * S_{rp}(\mathcal{RP}) + W_{wp} * S_{wp}(\mathcal{WP}) +$$
$$W_m * S_m(\mathcal{M}) + W_{da} * S_{da}(\mathcal{DA}) +$$
$$W_{di} * S_{di}(\mathcal{DI}) + W_A * S_a(\mathcal{A})$$

Figure 6: Ranking Model – Formulae for measuring the impact of APs.

	W_{rp}	W_{wp}	W_m	W_{da}	W_{di}	W_a
C1	0.7	0.15	0.05	0.04	0.02	0.02
C2	0.4	0.4	0.1	0.04	0.02	0.02

(a) Ranking model configurations

	S_{rp}	S_{wp}	S_m	S_{da}	S_{di}	S_a
Index Underuse	1.5x	0	0	0	0	0
Enumerated Types	0	>10x	2	1	0	0

(b) Impact of APs

Figure 7: Ranking Model Configurations – Illustration of the impact of the ranking model configuration on the ordering of APs.

types of frequently-observed queries in the presence and absence of each AP. For this analysis, we focus on the following types of queries: (1) a lookup query that retrieves a set of records from a table based on a highly selective predicate (SELECT), (2) an aggregation query that computes the sum of all the elements in a column (SUM), (3) a join query that combines the records in two tables in an application based on a join predicate (JOIN)), and (4) an update statement that modifies a set of records in a table based on a highly selective predicate (UPDATE). ap-rank uses the results of this quantitative analysis to estimate the potential speedup in executing the target application's queries by fixing an AP. For example, fixing the multi-valued attribute AP (Example 1) accelerates lookup and join queries by 636× and 256×, respectively.

❷ **Maintainability (M):** The next metric appraises the impact of each AP on the maintainability of the application. We conduct a qualitative analysis of the *number of changes* (C) needed in an application to support a new task in the presence and absence of each AP. This determines the degree of refactoring necessitated by the AP. If C is linearly or super-linearly dependent on the *number of queries* (Q) present in the application, then ap-rank will prioritize this AP. In contrast, a design wherein C is independent of Q improves the extensibility of the application.

Consider the enumerated types AP (Example 4). If the developer would like to rename a particular Role (*e.g.*, R2 $\mapsto$ R5), they would need to execute the following queries:

```
ALTER TABLE User DROP CONSTRAINT IF EXISTS User_Role_Check;
UPDATE User SET Role='R5' WHERE Role='R2';
```

Figure 5b illustrates an alternate design wherein only one query (*i.e.*, $C = O(1)$) is sufficient for this refactoring.

```
UPDATE Role SET Role_Name='R5' WHERE Role_Name='R2';
```

❸ **Data Amplification (DA):** The next metric appraises the impact of APs on *data amplification*. AP-free design can shrink the storage footprint of an application.

Consider the enumerated types AP in GlobaLeaks (Example 4). The Role column can take the following STRING values: (R1, R2, and R3). Repeatedly storing these STRING values increases the storage footprint of the application. The alternate design presented in Figure 5b addresses this limitation by introducing a ROLE table and encoding the Role column in the USER table using INTEGER values: (1, 2, and 3). In addition to reducing data amplification, it allows the developer to utilize foreign key constraints (*e.g.*, to ensure that every user can only take on one of these roles) [5].

❹ **Data Integrity (DI):** The third metric characterizes the impact of each AP on data integrity. We examine how an AP affects the integrity of the application. For instance, consider the multi-valued attribute AP in GlobaLeaks (Example 1). If a user with User_ID u1 is deleted from the USER table, we need to manually execute another query to delete the u1 string from the comma-separated User_IDs field.

```
UPDATE Tenants SET User_IDs = REPLACE(User_IDs, ',u1', '')
WHERE User_IDs LIKE '%u1%';
```

If this query is not executed, the data integrity constraint will be violated. In contrast, if the database contains an intersection table as shown in Figure 2c, we can leverage DBMS features for preserving integrity constraints (*e.g.*, cascaded deletes as shown in §2.1).

❺ **Accuracy (A):** This metric characterizes the impact of the AP on the *accuracy* of the returned results. Consider the no foreign key AP (Example 3). In the QUESTIONNAIRE table, since there is no foreign key linking the Tenant_ID columns in the Tenant and QUESTIONNAIRE tables, delete operations will not be cascaded. The resultant dangling references lead to tuples with NULL values when these tables are joined.

5.2 Model for Ranking Anti-Patterns

We now present the ranking model that ap-rank uses for ordering the APs identified by ap-detect. Our goal is to prioritize the attention of developers on high-impact APs. The model leverages the metrics presented in §5.1.

---

[5]This relationship can also be preserved using a CHECK constraint [34]. However, this feature reduces performance and maintainability (§8.2).

`ap-rank` sorts the `APs` in the application in decreasing order of their estimated impact on performance, maintainability, and accuracy. To do this estimation, it maps the queries in the application to the standard types of queries that have already been evaluated. It then generates a query-aware ranking of `APs` in the application. The developer can tailor the weights used by the model for these different features: performance, maintainability, and accuracy. It then sends the ordered list of `APs` to `ap-fix`. For `APs` with multiple candidate fixes, `ap-fix` suggests the best fix based on the collection of queries present in the application. We defer a discussion of `ap-fix` to §6. As new performance data is collected over time, we update the ranking model to improve the quality of its decisions.

Model Components: The model consists of two components: (1) *intra-query* and (2) *inter-query* ranking components. The intra-query component ranks the `APs` detected in each query. It first computes the following metrics for each `AP`: (1) read performance ($\mathcal{RP}$), (2) write performance ($\mathcal{WP}$), (3) maintainability ($\mathcal{M}$), (4) data amplification ($\mathcal{DA}$), (5) data integrity ($\mathcal{DI}$), and (6) accuracy ($\mathcal{A}$).

It then aggregates these metrics using the weights shown in Figure 6. The developer can configure these weights to best meet their applications requirements. For instance, if an application requires higher read performance, the developer can increase the *read performance* weight. `ap-rank` uses the computed aggregate score for ranking the `APs` within a query and to compute the score for each `AP`.

The inter-query component sorts `APs` based on their impact on all the queries in the application. The developer can choose one of two inter-query ranking models: ❶ based on number of `APs` in each query (*i.e.*, queries with more `APs` are ranked higher), or ❷ based on the computed score.

Example 6: Consider a query suffering from the *index underuse* and *enumerated types* `APs`. Figure 7a illustrates two different configurations of the ranking model (*C1* and *C2*). Figure 7b lists the metrics associated with the detected `APs`. The first configuration (*C1*) prioritises read performance ((*e.g.*, analytical workloads). So, it ranks the index underuse `AP` (score = 0.21) higher than the enumerated types `AP` (score = 0.175). In contrast, the second configuration (*C2*) gives equal priority to both read and write performance (*e.g.*, hybrid transactional/analytical workloads). So, it ranks the enumerated types `AP` (score = 0.47) higher than the index underuse types `AP` (score = 0.12). In this manner, `ap-rank` allows the developer to prioritise `APs`.

Conflicting Fixes: Fixes for `APs` detected in an application may conflict with each other. `ap-rank` assists the developer in resolving these conflicts by prioritising the `APs`. For instance, consider an application with these two `APs`:

Too Many Joins and Enumerated Types. To resolve the latter `AP`, the developer must create a new table for the attribute with an enumerated type (*e.g.*, `ROLE` table). However, this fix would increase the performance impact of former `AP` as it would require the developer to connect the newly added table with an additional `JOIN` in `SELECT` queries. sqlcheck orders the detected `APs` based on the user-specified ranking model. So, if the developer is prioritising read performance, then they may fix the former `AP` first and ignore the latter one. In this manner, they can iteratively fix the `APs` in the application based on their impact score from `ap-rank`.

6 Fixing Anti-Patterns

Merely identifying the high-impact `APs` will not be sufficient since application developers who are experts in other domains are likely not familiar with anti-patterns [24, 39]. `ap-fix` addresses this problem by automatically suggesting alternate database designs and queries that are tailored to the application. We begin with an overview of the algorithm used by `ap-fix`. We then describe the query repair engine that `ap-fix` leverages for rewriting SQL queries in §6.1.

Overview: As shown in Algorithm 4, `ap-fix` takes the following inputs: (1) a list of detected `APs`, (2) the parse trees of the queries containing those `APs`, and (3) the context of the application. Depending on the types of `APs`, it fetches the associated rules for fixing them. Besides targeting the queries containing the `APs`, `ap-fix` retrieves the list of queries $\mathcal{I}$ that are also impacted by the `AP` fix from the application's context. It appends these impacted queries to the list of queries containing `APs` to construct the list of queries that must be transformed ($\mathcal{Z}$). `ap-fix` then passes this list to the *query repair* engine (Line 7). The rule engine checks whether it can generate non-ambiguous query transformations for a given query based on the `APs` that it contains. If that is the case, then it applies those transformations on the query's parse tree (Line 9). It then transforms the parse tree to a SQL string based on the dialect used by the application. If it cannot generate non-ambiguous transformations, then it returns a textual fix that is tailored based on the context (Line 12). The application developer must subsequently follow the guidance provided in the textual fix to manually resolve the detected `APs`.

6.1 Query Repair Engine

The query repair engine transforms a given SQL statement based on a set of *rules* for fixing `APs`. The rule system is instrumental in facilitating our experimentation with statement transformations for two reasons. First, the rule system paradigm makes it easy for `ap-fix` to exploit the complicated triggering interactions between the repair rules, thereby obviating the need for explicitly laying out the flow of control between rules. Second, the rule system is extensible. This

Algorithm 4: Fixing Anti-Patterns

```
    input  : detected anti-patterns 𝒫, parsed queries 𝒬, context 𝒞
    output : anti-pattern fixes ℱ
 1  fixes ℱ ← { }
 2  for anti-pattern p in 𝒫 do
 3  │   fix rules ℛ ← GetRulesForAntiPattern(p)
    │   // Identify queries which are impacted by the anti-pattern fix
 4  │   impacted-queries I ← GetImpactedQueries(p, 𝒞)
 5  │   to-be-transformed-queries 𝒵 ← 𝒬 ∪ I
    │   // Pass to-be-transformed queries to the query repair engine
 6  │   for query z in 𝒵 do
 7  │   │   query transformations 𝒯 ← GetTransformations (z, p)
 8  │   │   if 𝒯 is not empty then
 9  │   │   │   transformed-parsed-query t ← Transform(z, 𝒯)
10  │   │   │   fixed-sql-query f ← ToSql(t)
11  │   │   else
    │   │   │   // Return a textual fix tailored for application
12  │   │   │   textual fix f ← GetTextualFix(p, z)
13  │   │   ℱ.append(f)
14  return ℱ
```

extensibility allowed us to formulate and evaluate tens of transformations over time.

In addition to rewriting existing SQL statements, `ap-fix` also needs to construct new statements for certain `APs`. For example, in the case of the multi-valued attribute `AP` (1), `ap-fix` first constructs a new `HOSTING` table and then updates the schema of the `TENANTS` table, as shown below:

```
/* Create an intersection table */
CREATE TABLE Hosting (
    User_ID VARCHAR(10) REFERENCES Users(User_ID),
    Tenant_ID VARCHAR(10) REFERENCES Tenants(Tenant_ID)
);
/* Drop redundant column */
ALTER TABLE Tenants DROP COLUMN User_IDs;
```

A key challenge for the query repair engine is that it must identify all the queries which are impacted by the anti-pattern fix and transform them as well. For instance, with the intersection table, `ap-fix` rewrites the query for retrieving information about the users served by tenant thus:

```
/* Retrieve information about users served by tenant */
SELECT * FROM Hosting as H JOIN Tenants as T
ON H.User_ID == T.User_ID WHERE H.Tenant_ID = 'T1';
```

Rule Representation: We represent rules in our engine as pairs of functions in a procedural language. Each rule consists of: (1) a *detection function* (§4), which does an arbitrary check and sets a flag TRUE or FALSE, and (2) an *action function*, which, if the condition function sets the flag TRUE, is invoked to take an arbitrary action, such as transforming existing SQL statements and creating new SQL statements.

7 Implementation

SQLCHECK is implemented in Python [35] and exports the following three interfaces. (1) Interactive Shell, (2) REST, and (3) GUI. Application developers and SQL IDE developers can 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 [36]) and directly use the interactive shell interface to execute SQL queries or leverage these sub-modules in other tools.

  ```
  # Import the anti-pattern finder method
  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 [32].

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

- **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 [22].

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.

8 Evaluation

We evaluated SQLCHECK on a variety of real-world SQL queries to quantify its efficacy in detecting, ranking and fixing `APs`. We illustrate that:

- **Detection:** SQLCHECK detects a wider range of `APs` (26) in real-world DBMS applications compared to DBDEO (11). `ap-detect` has **48%** fewer false positives and **20%** fewer false negatives than DBDEO resulting in higher precision and recall. SQLCHECK found 32 major `APs` in 15 real-world web applications.

- **Ranking:** `ap-rank` allows the developer to order `APs` based on their impact. Its ranking model is derived through an empirical analysis of Globaleaks. We show

AP Name	S	D	Both	TP-S	FP-S	TP-D	FP-D
Pattern Matching	1037	524	28	705	332	0	524
God Table	27	170	3344	27	0	0	170
Enumerated Types	414	42	48	411	3	0	43
Rounding Errors	352	7	1074	329	23	0	7
Data in Metadata	18	584	1226	18	0	93	491
Adjacency List	0	10	93	0	0	0	10
Total:	1848	1337	5813	1588	358	93	3783

Table 2: Detection of Anti-Patterns: Comparison of the number of APs identified by SQLCHECK and DBDEO in the query benchmark. Columns **S** and **D** report the APs detected by *only* that tool. Column **Both** lists the APs detected by both tools. **TP** and **FP** refer to true and false positives, respectively.

that the average and maximal impact of APs on runtime performance is 4× and >10000×, respectively.

- **Fixing:** We conduct a user study to validate the efficacy of SQLCHECK in fixing APs. SQLCHECK's efficacy in resolving the APs in the queries written by users is 51%. Overall, the participants of the study confirmed that SQLCHECK helped eliminate APs in their applications.

8.1 Detection of Anti-Patterns

To our knowledge, DBDEO is the closest system to SQLCHECK. DBDEO is effective in uncovering APs in real-world SQL queries. However, DBDEO's query analysis algorithm suffers from low precision and recall. Furthermore, it differs from SQLCHECK in that it neither ranks the APs based on their impact nor suggest fixes for the detected APs. In this experiment, we compare the AP-coverage and accuracy of SQLCHECK against that of DBDEO.

QUERY BENCHMARK: We download 1406 open-source repositories containing SQL statements from GitHub [23]. We extract around 174 thousand string-quoted embedded SQL statements from these repositories. We then use regular expressions to extract SQL statements from the files contained in these repositories. We evaluate SQLCHECK under two different configurations: (1) with only intra-query analysis, and (2) with both intra- and inter-query analyses. The open-source applications hosted on GitHub only contain SQL queries and not their associated databases. So, SQLCHECK cannot leverage its *data analysis* techniques in this experiment.

RESULTS: We group and aggregate the detected APs based on their type. The results (details in Table 3) are as follows:

- DBDEO detects 14764 APs (11 types of APs).

- SQLCHECK (only intra-query analysis) detects 86656 APs (18 types of APs).

- SQLCHECK (intra- and inter-query analysis) detects 63058 APs (21 types of APs).

- **Coverage and Accuracy:** Under both configurations, SQLCHECK detects a wider range of APs compared to DBDEO. With only intra-query analysis enabled, SQLCHECK

finds 2.6× more APs than DBDEO. We attribute this increase in recall to two factors. First, SQLCHECK supports 26 types of APs (DBDEO only supports 11 types.) Second, SQLCHECK uses detection rules that are capable of uncovering different variants of the same AP (*e.g.*, set of regular expressions for identifying Multi-Valued Attribute: `(id\\s+regexp)|(id\\s+like)`). This increase in recall also results in higher false positives (*i.e.*, lower precision). Enabling both intra- and inter-query analyses mitigates this problem. Under this configuration, SQLCHECK reports three additional types of AP but 1.8 × fewer APs compared to the prior configuration. This is because it eliminates false positives by leveraging the inter-query context.

- **Dialect-Coverage:** Qualitatively, both SQLCHECK and DBDEO support a wide range of SQL dialects. We attribute this to their usage of `sqlparse`, a non-validating parser, that supports diverse dialects. Furthermore, SQLCHECK leverages `sqlalchemy` to construct the query and context objects in a DBMS-agnostic manner [42].

We next conduct a manual analysis of the APs reported by DBDEO and SQLCHECK in the query benchmark for a subset of APs. We do not examine certain AP (*e.g.*, No Primary Key, Index Underuse) because DBDEO does not report the query in which the AP was detected. The results are shown in Table 2. SQLCHECK has 48% fewer false positives and 20% fewer false negatives compared to DBDEO. This illustrates the impact of intra- and inter-query analyses in increasing the precision and recall of AP-detection.

8.2 Ranking and Repair of Antipatterns

We next examine the impact of APs on runtime performance. In particular, we compare the query execution time before and after fixing a given AP in a real-world application.

EXPERIMENT SETUP: We aggregate the APs detected in the query benchmark presented in §8.1 based on their associated application. We rank these applications based on the frequency and types of detected APs. Based on this ranking, we select the Globaleaks application for this experiment [17].

Globaleaks leverages `sqlalchemy` (an object-relational mapping (ORM) framework) [42]. We first transform the ORM operations to SQL queries. We then recreate the database schema on a DBMS instance (PostgreSQL v11.2) and load a synthetic dataset containing 10 M records (19 GB across 11 tables). Globaleaks inherently contains ten types of APs. We infuse three additional APs for quantifying their performance impact. For instance, we add comma-separated strings in a column to infuse the multi-valued attribute AP.

We quantify the performance impact of every AP. For each AP, we execute different types of queries in Globaleaks under two configurations: (1) before the AP is fixed, and (2) after the

AP is fixed using the feedback provided by `ap-fix`. For each query, we report the average execution time of five runs.

Index Overuse AP: This `AP` is associated with the creation of too many infrequently-used indexes. As shown in Figure 8a, the performance impact of this `AP` is significant for the `UPDATE` statement. We attribute this to the overhead of maintaining the indices. The update operation is 10 × slower when there are five indices on the field being updated. Thus, it is advisable to only create those indexes whose impact on query processing is significant. Another fix for this `AP` is to maintain a multi-column index as opposed to maintaining multiple single-column indices.

Index Underuse AP: Figure 8b illustrates the impact of not having important indices on columns. We execute a query which performs a post-grouping aggregation operation. The query execution time drops by 1.3× when we create an index on the column contained in the `GROUP BY` clause. This is because the index eliminates the overhead of the grouping operation. Figure 8c illustrates a scenario where fixing this `AP` reduces performance. We consider a scan query with a predicate on a column with low cardinality. The query executes 3× slower when it uses the index as opposed to a table scan. We attribute this to the low cardinality of the indexed column [12]. The query analysis rule incorrectly flags this query due to missing indices. sqlcheck eliminates this false positive by leveraging its data analysis rule which takes cardinality into consideration.

No Foreign Key Exists AP: Figures 8d to 8f illustrate the performance impact of this `APs` on an `UPDATE` statement and a scan query. The performance impact is not prominent in both cases since the PostgreSQL does *not* automatically create an index to maintain the foreign key constraint. An index explicitly constructed by the user accelerates the `UPDATE` operation by 142×. This `AP` also has a significant impact on maintainability and data consistency. This is because the foreign key constraint must be preserved by the developer using complex application-level logic. For instance, in case of a cascaded `DELETE` operation, a developer will need to issue two SQL statements to update the values in both the reference and referencing tables. Otherwise, the referential integrity constraint will not be preserved. This increases the complexity associated with maintaining the application.

Enumerated Types AP: With this `AP`, the developer uses the `CHECK` constraint feature of the DBMS [34]. We measure the time taken to update a value of the `Role` column covered by the constraint (R2 ↦ R5). This consists of three steps: (1) an `ALTER` operation to drop the `CHECK` constraint on the `Role` column, (2) updating the value using an `UPDATE` statement based on a predicate, and (3) an `ALTER` operation to add the constraint back onto the column. In contrast, if the database contains an intersection table as shown in Figure 2c, the same task can be accomplished using an `UPDATE` statement. Figures 8g to 8i illustrates the performance impact of this `AP`. Eliminating this `AP` improves performance by more than 1000× in case of `UPDATE` and `INSERT` operations. The impact is less prominent in case of the scan query due to the overhead of the `JOIN` operator. The `AP`-free design improves maintainability by reducing the number of queries required for performing a given task (*e.g.*, `Role` update). Furthermore, it reduces the storage footprint by reducing data duplication.

Severity of APs: Impact of an `AP` depends on the application context (*e.g.*, Enumerated Types `AP` will not affect performance if the attribute's domain does not change). Certain `APs` may stem from application requirements. For instance, it may not be possible to simplify a query with Too Many Joins. Lastly, a subset of `APs` will always have negative impact (*e.g.*, No Foreign Key will always affect data integrity).

8.3 User Study

In this experiment, we evaluate the efficacy of sqlcheck through a user study. We recruited 23 graduate and undergraduate students majoring in Computer Science with varying degrees of expertise in SQL.

Experiment Setup: We tasked the participants to construct a set of SQL queries for a bike e-commerce application. The requirements are twofold: (1) design a performant and extensible database design for this application, and (2) formulate performant SQL queries to support application features. We curated a set of sixteen features that are associated with one or more `APs` (*e.g.*, shopping cart, list of products).

The participants use the GUI Interface presented in §7. We track: (1) the original SQL queries developed by the user, (2) the fixes suggested by sqlcheck for the `APs` detected in the original queries, and (3) the re-formulated SQL queries developed by the user that incorporate these fixes. We also collect qualitative feedback from the participants about the accuracy and utility of the detected `APs` and their fixes.

Results: The participants constructed 987 SQL statements. sqlcheck detected and suggested fixes for 207 `APs`. Most of the participants (20 out of 23) took the 187 `APs` detected in their queries into consideration. They refactored the queries to resolve 96 `APs`. They ignored the remaining 91 fixes. The reasons for this are twofold: (1) ambiguous fixes (31 fixes), and (2) incorrect fixes (60 fixes) given the requirements of the application. Thus, the participants leveraged 51% of the fixes suggested by sqlcheck. If we also include the `APs` labeled as ambiguous from the participants' perspective, then the efficacy increases to 67%.

We compare the distribution of `APs` detected in the participants' queries using DBDEO and sqlcheck. The results are shown in Table 3. There is significant variation in the
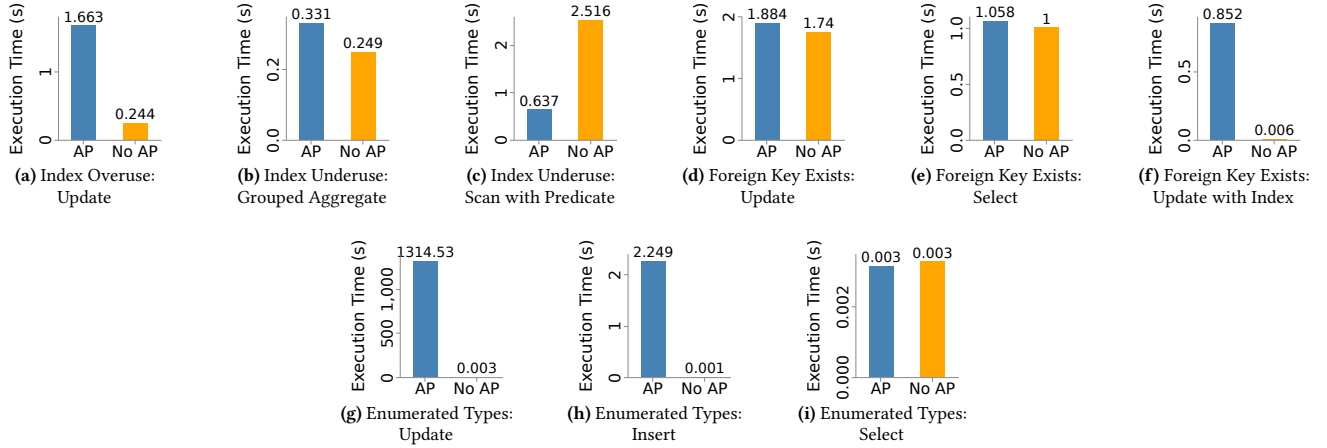
Figure 8: Ranking and Repair of AP: Performance impact of AP on different types of SQL statements.

frequency of APs. For instance, the No Primary Key AP is 14× more prevalent than the Pattern Matching AP.

We next examine the variance in SQL skills of the participants. The number of queries executed, detected APs, and accepted APs follow these distributions: (μ=42.5, Q2=46), (μ=9.35, Q2=8), and (μ=4.8, Q2=46). The high variance in SQL skills illustrates the need for an automated toolchain for detecting APs and suggesting fixes. The qualitative feedback from the participants indicate that they predominantly found sqlcheck to be helpful in understanding APs.

8.4 Web Applications & Databases

In this experiment, we first evaluate the efficacy of sqlcheck in finding, ranking, and suggesting fixes for APs in real-world web applications on GitHub. We apply sqlcheck on 15 actively-developed Django-based applications [1].

Experiment Setup: We first manually deploy each of these applications on PostgreSQL. We then collect the SQL queries either by running the integration tests or by manually interacting with the application. Lastly, we report the high-impact APs to the developers by either raising issues on GitHub or through the official developer forum. Before reporting the APs, we manually analyse them to study their significance based on the application-specific context. We order the APs impact metrics thus: read performance, maintainability, write performance, accuracy, and amplification.

Results: As shown in Table 4, sqlcheck detected 123 APs across these applications (Ref. Section B). We reported 32 APs based on their impact score and the application-specific context. We do not report low severity APs (*e.g.*, Generic Primary Key) and those that require a deeper understanding of application requirements (*e.g.*, Too Many Joins).

We have received responses from all but three of these development teams. Eleven teams acknowledged the existence of APs in their applications. They attribute these APs

Anti-Pattern	GitHub Rep		User Study		Kaggle	
	D	**S**	**D**	**S**	**D**	**S**
No Primary Key	628	6875	22	70	-	68
Column Wildcard Usage	0	12313	0	54	-	-
Data in Metadata	1907	1352	43	39	-	9
Enumerated Types	90	462	11	30	-	-
Index Underuse	82	506	40	30	-	-
God Table	3514	3371	22	28	-	-
Implicit Columns	0	26488	0	24	-	-
Readable Password	0	295	0	20	-	-
Clone Table	1990	516	21	12	-	-
Rounding Errors	1081	1426	91	10	-	-
Generic Primary Key	0	5123	0	8	-	25
Multi-Valued Attribute	2539	1503	3	6	-	20
Pattern Matching	552	1065	25	5	-	-
Adjacency List	103	93	0	0	-	-
No Foreign Key	0	1389	0	0	-	10
External Data Storage	0	63	0	0	-	-
Index Overuse	228	228	0	0	-	-
Concatenate Nulls	0	63	0	0	-	-
Ordering by Rand	0	27	0	0	-	-
Distinct and Join	0	4	0	0	-	-
Too many Joins	0	4	0	0	-	-
Missing Timezone	-	-	-	-	-	12
Incorrect Data Type	-	-	-	-	-	28
Denormalized Table	-	-	-	-	-	16
Information Duplication	-	-	-	-	-	1
Redundant Column	-	-	-	-	-	11
No Domain Constraint	-	-	-	-	-	0
Total:	14764	63058	278	336	-	200

Table 3: Distribution of APs– Distribution of APs detected by sqlcheck (S) and dbdeo (D) in queries collected from repositories on GitHub and written by the user study participants.

to the default behavior or lack of certain features in Django. Four teams are incorporating the fixes from sqlcheck. Three teams are looking for alternate fixes. Three teams did not share their course of action. One team decided not to fix the reported APs given their application-specific requirements. Most of these teams were interested in understanding the implications of these APs and requested us to send patches.

#	GitHub Repo	# AP Det	# AP Rep
1	Globaleaks	10	2
2	Django-oscar	12	2
3	Saleor	10	2
4	Django-crm	8	4
5	django-cms	11	1
17	**Total**	**123**	**32**

Table 4: Evaluation of SQLCHECK on Web Applications: The APs detected by SQLCHECK (# APs Det) in a subset of 15 Django applications. We list the major APs that we reported (# APs Rep).

#	Kaggle Database	# AP
1	The History of Baseball	41
2	Soccer Dataset	20
3	Acad. Research from Indian Univ.	17
4	Pesticide Data Program	13
5	Board Games	12
31	**Total**	**200**

Table 5: Evaluation of SQLCHECK on real-world databases: The APs detected by SQLCHECK in a subset of 31 Kaggle databases.

In one of these applications, SQLCHECK found APs that introduced by a third-party library. In another application, we found an existing issue related to the Too Many Joins AP. The developers found that replacing the ORM-generated query with a simpler, hand-written query greatly improved performance. This experiment illustrates the efficacy of SQLCHECK in assisting application developers in practice.

DATA ANALYSIS: We next evaluate the efficacy of SQLCHECK in finding APs in real-world databases on Kaggle [26]. We download 31 SQLite databases and apply the data analysis rules of SQLCHECK on them (§4.2). As shown in Table 5, SQLCHECK detects 200 APs across these databases (Ref. Section A). This experiment illustrates the efficacy of SQLCHECK in detecting APs by only analysing data (without queries).

8.5 Limitations And Future Work

ANTI-PATTERN COVERAGE, DISCOVERY, AND EVOLUTION: SQLCHECK currently detects 26 types of APs. We intend to add support for more known APs in the future. However, it is unclear how to automatically discover new types of APs in SQL queries. Furthermore, the performance impact of an AP can evolve over time. For instance, the performance impact of the *Adjacency list* AP was prominent in PostgreSQL v9 (5×). However, it is no longer significant (1.1×) in v11.

DIALECT COVERAGE AND QUERY REPAIR: SQLCHECK is designed to support multiple SQL dialects for higher utility. We accomplish this using a non-validating query parser (§4.1). However, it is infeasible to handle dialect-specific features, especially in complex queries. The usage of a non-validating query parser also restricts the set of queries wherein we can automatically rewrite the query to fix the AP. This is because we do not have enough syntactical information for query rewriting. We instead fall back on tailored textual

fixes in these scenarios. We made this decision to increase the utility of SQLCHECK. The data analyzer (§4.2) is built on top of SQLAlchemy so that it can support diverse DBMSs (*e.g.*, PostgreSQL, MySQL). Thus, the set of DBMSs that can be analyzed using `ap-detect` is constrained by those that are supported by SQLAlchemy.

9 Related Work

TRANSFORMING DATABASE APPLICATIONS: Although program analysis has a long history in software engineering, it has not been extensively studied by the DBMS community. Recent research efforts have focused on transforming database-backed programs to improve performance [9, 19, 45]. Ramachandra *et al.* present application transformations that enable asynchronous and batched query submission [37]. DBridge presents a set of holistic optimizations including query batching and binding, and automatic transformation of object-oriented code into synthesized queries [16, 37]. Cheung *et al.* describe techniques for batching queries to reduce the number of round trips between the application and database servers [7–10].

OBJECT-RELATIONAL MAPPING: Researchers have studied the impact of ORM on application design and performance [4–6, 43, 46]. Yang *et al.* perform a comprehensive study of performance issues in database applications using profiling techniques [47].

This paper is the first to explore the problems of automatically ranking and fixing APs in database applications.

10 Conclusion

In this paper, we presented 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 improves upon DBDEO, the state-of-the-art tool for detecting AP, by using 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. We discussed how SQLCHECK 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.

Acknowledgements

References

[1] [n.d.]. Django. https://www.djangoproject.com.

[2] Andi Albrecht. 2019. sqlparse: A non-validating SQL parser module for Python . https://github.com/andialbrecht/sqlparse.

[3] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "what-if" index analysis utility. *SIGMOD Record* 27, 2 (1998), 367–378. https://doi.org/10.1145/276305.276337

[4] Tse-Hsun Chen. 2015. Improving the quality of large-scale database-centric software systems by analyzing database access code. In *Proc. of ICDEW*. IEEE, 245–249.

[5] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proc. of ICSE*. ACM, 1001–1012.

[6] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2016. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Transactions on Software Engineering* 42, 12 (2016), 1148–1161.

[7] 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*.

[8] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. 2016. Sloth: Being lazy is a virtue (when issuing database queries). *ACM Transactions on Database Systems (TODS)* 41, 2 (2016), 8.

[9] 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.

[10] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. *In SIGPLAN* 48, 6 (2013), 3–14.

[11] Digital Equipment Corporation. 1992. SQL-92 Standard. https://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

[12] Ember Crooks. 2013. Why low cardinality indexes negatively impact performance. https://www.ibm.com/developerworks/data/library/techarticle/dm-1309cardinal/index.html.

[13] 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).

[14] Thomas H Davenport and DJ Patil. 2012. Data scientist. *Harvard business review* 90, 5 (2012), 70–76.

[15] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. 2005. Automatic Performance Diagnosis and Tuning in Oracle. In *CIdR*.

[16] K Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S Sudarshan. 2016. Extracting equivalent sql from imperative code in database applications. In *Proc. of SIGMOD*. ACM, 1781–1796.

[17] Hermes Center for Transparency and Digital Human Rights. 2019. GlobaLeaks. https://www.globaleaks.org/.

[18] Goetz Graefe. 1993. Query evaluation techniques for large databases. *In CSUR* 25, 2 (1993), 73–169.

[19] Ravindra Guravannavar and S Sudarshan. 2008. Rewriting procedures for batched bindings. *In Proceedings of VLDB* 1, 1 (2008), 1107–1123.

[20] E Iee. 1990. Ieee standard glossary of software engineering terminology. (1990).

[21] Cunningham & Cunningham Inc. 2014. C2 Wiki. http://wiki.c2.com/?AntiPatternsCatalog.

[22] Facebook Inc. 2019. ReactJS. https://reactjs.org.

[23] GitHub Inc. 2019. GitHub. https://github.com.

[24] Stitch Inc. 2018. The State of Data Science. https://www.stitchdata.com/resources/the-state-of-data-science/.

[25] Stack Exchange Inc. 2010. StackOverflow Wiki. https://stackoverflow.com/questions/346659/what-are-the-most-common-sql-anti-patterns.

[26] Kaggle. 2020. Kaggle. https://kaggle.com.

[27] Bill Karwin. 2010. *SQL antipatterns: avoiding the pitfalls of database programming*. Pragmatic Bookshelf.

[28] Sushil Kumar. 2003. Oracle Database 10g: The Self-Managing Database. White Paper.

[29] Eva Kwan, Sam Lightstone, Adam Storm, and Leanne Wu. 2002. *Automatic Configuration for IBM DB2 Universal Database*. Technical Report. IBM.

[30] David Lomet, Alan Fekete, Gerhard Weikum, and Mike Zwilling. 2009. Unbundling transaction services in the cloud. *arXiv preprint arXiv:0909.1768* (2009).

[31] Dushyanth Narayanan, Eno Thereska, and Anastassia Ailamaki. 2005. Continuous Resource Monitoring for Self-predicting DBMS. In *MASCOTS*. 239–248.

[32] Pallets. 2019 . Python-Flask. http://flask.palletsprojects.com/en/1.1.x.

[33] PostgreSQL. 2014. Postgres Query Parsing. https://wiki.postgresql.org/wiki/Query_Parsing.

[34] PostgreSQL. 2019. Constraints in PostgreSQL. https://www.postgresql.org/docs/current/ddl-constraints.html#DDL-CONSTRAINTS-CHECK-CONSTRAINTS.

[35] Python Software Foundation. 2019 . Python. https://www.python.org.

[36] Python Software Foundation. 2019. PyPi. https://pypi.org.

[37] 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.

[38] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database Management Systems* (3 ed.). McGraw-Hill, Inc., New York, NY, USA.

[39] Toby Segaran and Jeff Hammerbacher. 2009. *Beautiful data: the stories behind elegant data solutions*. " O'Reilly Media, Inc.".

[40] Microsoft SQL Server. 2019. Database Engine Tuning Advisor. https://docs.microsoft.com/en-us/sql/relational-databases/performance/database-engine-tuning-advisor?view=sql-server-ver15.

[41] 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.

[42] SQLAlchemy. 2019. SQLAlchemy. https://www.sqlalchemy.org/.

[43] 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.

[44] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*. 1009–1024.

[45] Cong Yan and Alvin Cheung. 2016. Leveraging lock contention to improve OLTP application performance. *In Proceedings of VLDB* 9, 5 (2016), 444–455.

[46] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *Proc. of CIKM*. ACM, 1299–1308.

[47] 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.

Appendix A Data Analysis

In this experiment, we apply SQLCHECK's data analysis rules on 31 publicly-available SQLite databases from Kaggle. Table 6 lists the SQLite databases from Kaggle [26] that we use in this experiment along with the `APs` detected in these databases. We found `200 APs` across 31 databases using SQLCHECK. The results of this experiment are discussed in §8.4.

Appendix B Real World Applications

We tested SQLCHECK on 15 actively developed, open-source applications based on the Django ORM [1]. In this experiment, we pass the SQL queries obtained from either running the application's integration tests or by manually interacting with the application to SQLCHECK. We rank the detected `APs` based on their impact and reported the ones with high impact. The `APs` detected are summarized in Table 7. The results of this experiment are discussed in §8.4.

Appendix C Database Tuning Advisors

There is a large body of work related to automated database administration tools geared towards optimizing: (1) database physical design, and (2) database knob configuration.

The former set of tools focus on selecting the best physical design (*e.g.*, materialized views) for a given database that optimizes the target metrics (*e.g.*, latency) and satisfies the budget constraints (*e.g.*, hardware resources). These tools include Microsoft's AutoAdmin [3] and DETA [40] The latter set of tools instead focus on selecting the best knob configuration (*e.g.*, buffer pool size) for a given DBMS. These tools include OtterTune [44] and vendor-specific offerings [15, 28, 29, 31].

Table 8 compares the core features of SQLCHECK and a tuning advisor (DETA). SQLCHECK complements DETA in that it focuses on other aspects of database applications besides physical design (*e.g.*, logical design `APs` and query `APs`). We recommend the user to leverage the accurate feedback of the physical design tuning tool to fix the detected physical design `APs`, since it leverages the cost model of the query optimizer. SQLCHECK assists application developers in preventing `APs` during the application development phase by suggesting better alternatives (as opposed to tuning tools that are more often used post deployment).

#	SQLite Database	# AP	Detected Anti-Patterns
1	Board Games	12	No Primary Key, Data in Metadata, Incorrect Datatype
2	Pennsylvania Safe Schools Report	1	No Primary Key
3	Soccer Dataset	20	Generic Primary Key, Data in Metadata, Missing Timezone, Multivalued Attribute
4	SF Bay Area Bike Share	11	No & Generic Primary Key, Incorrect Datatype, Missing Timezone, Denormalized Table
5	US Baby Names	2	Generic Primary Key
6	Pitchfork Music Data	10	No Primary Key, Missing Timezone, Information Duplication, Denormalized Table
7	Acad. Research from Indian Univ.	17	No Primary Key, Incorrect Datatype, Redundant Column, Multivalued Attribute
8	What.CD HipHop	3	No Primary Key, Multivalued Attribute
9	Snap Meme-Tracker	1	Missing Timezone
10	NIPS papers	4	Generic Primary Key, Denormalized Table
11	US Wildfires	2	No Primary Key, Redundant Column
12	Que from crossvalidated StackExc	3	No Primary Key
13	The History of Baseball	41	No Primary Key, Data in Metadata, Incorrect Datatype, Multivalued Attribute
14	Twitter US Airline Sentiment	2	Denormalized Table
15	Hilary Clinton Emails	8	Generic Primary Key, Incorrect Datatype
16	SEPTA - Regional Rail	2	Incorrect Datatype, Missing Timezone
17	US Consumer finance Complaints	9	No Primary Key, Incorrect Datatype, Multivalued Attribute, Denormalized Table
18	1st GOP Debate Twitter Sentiment	1	Generic Primary Key
19	SF Salaries	2	Generic Primary Key, Denormalized Table
20	Freight Matrix Transportation	5	No Primary Key, Data in Metadata, Redundant Column
21	WDIdata	9	No Primary Key, Multivalued Attribute
22	Amazon Movie Reviews Dataset	2	No Primary Key, Multivalued Attribute
23	UK Arms Export License	3	No Primary Key
24	Amazon Fine Food Reviews	1	Generic Primary Key
25	Stackoverflow Question Favourites	1	Multivalued Attribute
26	Iron March	1	Redundant Column
27	C# Methods with Doc. Comments	4	Generic Primary Key
28	Pesticide Data Program	13	No Primary Key, Incorrect Datatype, Redundant Column
29	Monty Python Flying Circus	4	No Primary Key, Missing Timezone, Denormalized Table
30	Twitter Conv. about Black Panther	0	-
31	2016 US Election	6	No Primary Key, Data in Metadata, Denormalized Table
	Total	**200**	

Table 6: Data Analysis Results A list of SQLite databases from Kaggle [26] with the APs detected by applying the data analysis rules in SQLCHECK.

GitHub Repo	Stars	Contr.	Domain	# AP	APs Reported	R	A
Globaleaks	741	22	Whistleblower	10	2 (No Foreign Key, Enumerated Types)	✓	✓
Django-oscar	4.1k	217	E-commerce	12	2 (Rounding Errors, Index Overuse)	✓	✓
Saleor	6.5k	139	E-commerce	10	2 (Multivalued Attribute, Index Overuse)	✓	✓
Django-crm	654	17	CRM	8	4 (Index Underuse, Index Overuse, Pattern Matching, No Domain Constraint)	✓	✓
django-cms	7.2k	398	CMS	11	1 (Index Overuse)	✓	✓
wagtail-autocomplete	41	7	Utility	1	1 (Pattern Matching)	✓	✓
shuup	1.1k	41	E-commerce	6	1 (Index Overuse)	✓	✓
Pretix	821	113	E-commerce	11	3 (Index Overuse, Pattern Matching, No Domain Constraint)	✓	✓
Django-countries	755	35	Library	1	1 (Multivalued Attribute)	✓	✓
micro-finance	55	8	Finance	8	4 (Index Underuse, Index Overuse, Pattern Matching, No Domain Constraint)	✓	✓
bootcamp	1.9k	24	Social Ntwrk	5	1 (Index Overuse)	✓	✓
NetBox	6.2k	118	DCIM	9	3 (Index Overuse, Pattern Matching, No Domain Constraint)	✓	✓
Ralph	1.3k	43	Asset Mgmt	12	3 (Index Overuse, Pattern Matching, No Domain Constraint)	✓	✕
Tiaga	6.5k	139	E-commerce	9	2 (Index Overuse, No Domain Constraint)	✓	✕
wagtail	8.4k	397	CMS	10	2 (Index Overuse, No Domain Constraint)	✓	✕
Total				**123**	**32**		

Table 7: Evaluation of SQLCHECK on real-world applications on GitHub: A list of open-source database applications on GitHub along with their popularity and domain, the APs (counts and names) detected by SQLCHECK, whether these APs have been reported **(R)** as GitHub Issues and acknowledged **(A)** by the contributors.

Supported Features	DETA	SQLCheck
Index creation/destruction suggestions	✓	✓
Type of index to create based on workload	✓	✗
Materialized view creation/destruction suggestions	✓	✗
Suggestions tailored based on hardware constraints, workload, & data distribution	✓	✗
Table partitioning suggestions	✓	✗
Column type suggestions based on data	✗	✓
Query refactoring suggestions	✗	✓
Alternate logical schema design suggestions	✗	✓
Logical errors that may invalidate data integrity	✗	✓

Table 8: SQLCheck v/s Microsoft DETA [40] Comparison of the core features of sqlcheck against a physical design tuning advisor.