Cleaning Antipatterns in an SQL Query Log

Natalia Arzamasova, Martin Schäler, and Klemens Böhm

Abstract— Today, many scientific data sets are open to the public. For their operators, it is important to know what the users are interested in. In this paper, we study the problem of extracting and analyzing patterns from the query log of a database. We focus on design errors (antipatterns), which typically lead to unnecessary SQL statements. Such antipatterns do not only have a negative effect on performance. They also introduce bias on any subsequent analysis of the SQL log. We propose a framework designed to discover patterns and antipatterns in arbitrary SQL query logs and to clean antipatterns. To study the usefulness of our approach and to reveal insights regarding the existence of antipatterns in real-world systems, we examine the SQL log of the SkyServer project, containing more than 40 million queries. Among the top 15 patterns, we have found 6 antipatterns. This result as well as other ones gives way to the conclusion that antipatterns might falsify refactoring and any other downstream analyses.

Index Terms— SQL log analysis, patterns and antipatterns, data preprocessing

1 INTRODUCTION

NOWADAYS, various databases from different scientific domains are publicly available. They typically offer interfaces for declarative access, i.e., can be accessed in a very broad variety of ways. For the operators of such databases, it is very important to know what the users are interested. However, due to the public availability of the database, its owners cannot interact with all users to learn their interests. On the other hand, the queries issued by a user are a formal representation of his information needs. In other words, a query log is a perfect source of information to that end. However, analyzing such a log is difficult.

For example, [1] describes an approach to detect user interests based on the query log. They cluster queries, using the overlap of the data space accessed as the distance measure. In their case study with SkyServer, there were several clusters that domain experts could not explain. The queries in these clusters filter data by internal IDs. These attributes do not have any meaning in astronomy. We conclude that those SQL statements are follow-up queries of previous ones, i.e., they need to be considered in a context. Such follow-up queries appear to be frequent in the log. They introduce negative effects, e.g., falsify analyses, as we are about to explain. In software engineering, such an actuality is named antipattern [2]. An antipattern is a special case of a pattern [3]: while a pattern is a common solution, an antipattern is a pattern with a negative effect.

Example 1. Table 1 lists a sequence of SQL queries of a user. These statements reflect specific intentions of the user, i.e., form patterns. The second, the third, and the fourth query filter the tables using the same constant. Without the first query, one cannot understand this constant. Put differently, a join is computed outside of the database. This is an occurrence of the Circuitous Treasure Hunt (CTH) antipattern [4]. Next, the second and the third query select different columns of the same table. This is the Stifle antipattern [5].

TABLE 1
A series of statements from one user

#	Statements	Result
1	`SELECT E.empId FROM Employees E WHERE E.department = 'sales'`	12
2	`SELECT E.name, E.surname FROM Employees E WHERE E.id = 12`	John, Doe
3	`SELECT E.birthday, E.phone FROM Employees E WHERE E.id = 12`	12.03.1985, 01259863448
4	`SELECT count(orders) FROM Orders OWHERE O.empId = 12`	36

Example 2. Suppose that one wants to find hotspots of user interests. Queries 2 and 3 by the same user refer to the same data object, and a naive log-analysis scheme would count two occurrences of interest in this object. But it should not be overly controversial that these queries represent the same information need, at least when being issued right after each other. In other words, an occurrence of the Stifle has falsified this analysis.

Example 3. Consider again Table 1. Queries 2 to 4 can only be understood together with Query 1. This is because the Attribute id does not have any meaning from the domain perspective. Thus, if queries are rewritten with antipatterns removed, the specific user interest would be more obvious:

```
SELECT E.empId, E.name, E.surname,
```


- *N. Arzamasova is with the Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany. E-mail: natalia.arzamasova@kit.edu.*
- *M. Schäler is with the Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany. E-mail: martin.schaeler@kit.edu.*
- *K. Böhm is with the Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany. E-mail: klemens.boehm@kit.edu.*

***Please provide a complete mailing address for each author, as this is the address the 10 complimentary reprints of your paper will be sent*

*Please note that all acknowledgments should be placed at the end of the paper, before the bibliography (**note that corresponding authorship is not noted in affiliation box, but in acknowledgment section**).*

xxxx-xxxx/0x/$xx.00 © 200x IEEE Published by the IEEE Computer Society

```
E.birthday, E.phone, O.oCount
FROM Employees E INNER JOIN
(SELECT empId, count(orders) as oCount
FROM Orders GROUP BY empId) O
ON O.empId = E.empId
```

The topic of this paper is the detection of antipatterns in an SQL query log. We have consciously decided to see this as a general preprocessing step in the data-analysis processing chain, i.e., subsequent log-analysis tasks like user-interest identification or association-rule mining are decoupled from it. This is in line with the general, commonly accepted perspective that there is a data cleaning step that precedes more abstract, goal-oriented analyses. Having said this, we nevertheless have conducted combined experiments which reveal the usefulness of the preprocessing proposed in this article, see Section 6.9 below. Of course, with any data cleaning, an analyst needs to be aware of the fact that the cleaning has modified (biased) the original data. But that decoupling gives way to more flexibility as well as effectiveness and efficiency of the subsequent analyses. In other words, the approach described here is a variant of data cleaning, and we use the terms cleaning or solving for those rewrites.

A common method to detect antipatterns [4] requires access to the software that generates the requests. Regarding SQL antipatterns, this means that one would need to have access to all systems working with the database. This is practically impossible, for databases on the Web in particular. That solution also does not help regarding antipatterns in an already existing query log. As Example 1 has insinuated, one challenge when looking for SQL antipatterns is the identification of dependencies among subsequent queries. At first sight, a promising approach is re-querying. For instance, to know for sure that Statements 2, 3 and 4 depend on Statement 1, one should run the first statement again and inspect the result. However, this is not viable, for the following reasons:

1. Performance aspect: Re-running a significant part of a SQL log implies a huge load on the database.
2. Side effects aspect: The database will save these 're-run' queries in the query log; this will bias any subsequent log analysis.
3. Data persistence aspect: In the presence of modifications of the data set, the result of a re-issued query does not have to be the same as the original one.
4. Schema modification aspect: Because of database-schema refactoring such as renaming of attributes, old requests might even cause errors.

So when it comes to the design of a method that detects patterns and solves antipatterns, we see a difficulty in deciding what exactly should be rewritten.

Example 4. Consider Table 1 once more. If we did not have the information that all queries are from the same user, or if the attribute had a meaning in a specific domain, in contrast to 'id', it would be less clear if this were indeed occurrences of CTH and the Stifle.

This suggests to first examine how distinct/how frequent the clear cases are, and how far respective solutions will take us. It also is unclear which antipatterns one should focus on.

In this article we give answers to these questions by means of an empirical study based on a large, freely available query log. Our main steps and the core insights are as follows:

1. We provide formal definitions classifying a query load into normal queries, patterns, and antipatterns.
2. We describe our solution to detect and classify patterns and antipatterns as well as to solve antipatterns within a query log. While we confine ourselves to the CTH and the Stifle, we have designed a processing framework that can also accommodate other antipatterns (see Section 5.4).
3. Our empirical study relies on the log of the SkyServer system, covering a time span of 7 years and containing nearly 42 million queries.
4. In line with other research on data cleaning, our core evaluation criterion is the plausibility of our results (in contrast to result quality of any downstream analyses). For instance, the share of antipatterns in the SkyServer log is significant (6 antipatterns among the 15 most frequent patterns), and after removing them, all patterns among the 40 most frequent ones do represent meaningful information needs.
5. We present evidence that the results of previous studies of the SkyServer log (e.g., [6], [1]) would have been different, had the log been cleared of antipatterns.

Paper outline: Section 2 reviews processing query log and some well-known database antipatterns. In Section 3 we focus on SQL antipatterns which affect a query log. Section 4 provides definitions of pattern and antipatterns for our context, as well as rewrite rules whenever possible. In Section 5, we describe our framework in detail. Section 6 features the empirical study. Section 7 concludes.

2 RELATED WORK

In this section, we first look at work targeting query logs. Then, we review approaches to detect and remove antipatterns introduced by interaction with a database. We also refer to related work in other sections of this article, and this current section only deals with the relatively few related approaches not addressed elsewhere.

2.1 Processing Query Logs

Query-log analysis currently is a field of intensive study. An elementary distinction is between web logs and SQL logs. Studies on web-log processing such as [7] and [8] tend to focus on understanding of the user behavior through their information-seeking activities. [7] studies web search engine optimization by mining past queries. [8] proposes a context-aware query recommendation approach by mining click-through and session data.

Studies of SQL logs mainly consider publically available scientific databases. [1] analyzes the Sky Server log with the aim of identifying user interests within the data space. [6] uses the same use case to provide a recommen-

dations system for SQL queries. [9], [10] and [11] are detailed reports of the Sky Server user activities. They analyze both types of logs, SQL and web. The study provides various statistics regarding the first five [9] and ten years [10], [11] since SDSS SkyServer has gone online.

Another promising branch of SQL log analysis is diagnosing and repairing data errors caused by erroneous updates. [12] works with a log of update queries, UPDATE, INSERT and DELETE statements, and a set of known data errors to find and fix mistakes within a dataset. The processing of DML queries does not address our specific problem – finding antipatterns. Nevertheless, this study bears a connection to our work since it provides formal definition rules for discovering and solving the errors. Overall, query log analysis has different research threads, namely query recommendation, understanding user behavior and diagnosing errors. Our study relates to all these objectives. Finding and fixing antipatterns in a query log is a preprocessing step for further analysis, such as providing query recommendations [6] or investigating user behavior [1] without bias.

2.2 Review of Database Antipatterns

In the following, we briefly review research on database antipatterns. [13] lists semantic errors in SQL queries. Their insights stem from their experience while correcting database exams. Some of the mistakes listed lead to syntax errors. This work is orthogonal to ours. If a mistake is frequent, there might be a corresponding antipattern, which could be solved or removed from the log. [14] detects database *design* antipatterns by querying metadata tables. Such antipatterns reflect errors in the database schema, which is not the topic of our study. [15] proposes a framework to detect object-relational mappings (ORM) performance antipatterns. It is based on static code analysis and a rule-based approach. The detection of DML bug patterns is studied in [16]. DML queries, however, are not in the focus of our paper. The reason is that we aim to clean a query log of SELECT statements in order to facilitate further analyses on it (i.e., what do database users find interesting in the database).

3 FUNDAMENTALS

In this section we focus on two antipatterns, which influence analysis of an SQL query log. These are the Stifle [17] and the Circuitous Treasure Hunt [18]. They are also known to be the main reason for antipattern-related performance degradations. Our explanation includes suggestions for their detection and removal. We then point out limitations of current solutions, motivating a new approach.

3.1.1 Stifle Antipattern

The Stifle antipattern consists of several queries containing similar SQL statements [5]. The term *similar* does not have a formal definition. However, examples are queries being identical except for constants in the WHERE clause, as in Example 1. Processing such queries may be a bottleneck and has a negative impact on performance. When analyzing a query log, the Stifle may falsify results. For instance, as pointed out earlier, it blurs the representation of user interests.

Example 5. The following code generating SQL statements illustrates the Stifle antipattern. Thus, every `Id` in the `itemList` causes a request to Table T. In a log, the Stifle manifests itself as a sequence of similar statements.

```
for (int item: itemList)
{String sql =
"SELECT * FROM T WHERE Id = " + item;
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);}
```

A detection approach for the Stifle [17] is based on measurements of a *running* instance of software. It assumes that a high load is a high-probability indicator for the presence of antipatterns. A specific indicator for the Stifle is a large number of database calls per service and a small average number of result rows per query. Thus, this approach is based on general statistics and may not be precise enough. [18] proposes heuristics for measurement-based detection of several antipatterns on source-code level. There, the Stifle antipattern is characterized by many similar database requests. A Stifle is detected if two or more database requests from one user for a service exist. In addition, the requests need to have the same structure, except for the values passed to the method building or executing the query. Thus, this approach requires access to the source code of the service. In addition, it is based on comparing strings used in the source code which then form the query. In summary, these detection approaches are limited, and a more sophisticated approach having a more complete view on the actual query load is necessary.

We now review methods to rewrite instances of the Stifle antipattern. In software development this is called refactoring. [12] proposes *the Pack refactoring*. Their idea is to collect individual SQL statements and send them to the database in one batch.

Example 6. The Pack refactoring for Example 5 is:

```
String sql = "";
for (int item: itemList)
{sql = sql + "SELECT * FROM T WHERE Id = "
+ item + ";";}
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

After the Pack refactoring one gets a single request which consists of several SQL statements. This solution removes the unnecessary network overhead, for any future query. However, it still requires the same amount of database resources. It does not alter the query log. We for our part seek an approach that rewrites such queries in an existing log to facilitate meaningful analyses.

3.1.2 Circuitous Treasure Hunt (CTH)

The Circuitous Treasure Hunt (CTH) antipattern [19] has one similarity with the Stifle, as they both consist of several database requests. However, the individual CTH queries depend on each other. This means that a subsequent query requires the result of prior ones as input.

Similarly to the Stifle, a high database overhead is an

indication for the CTH antipattern [17]. However, in order to identify any CTH, either knowledge on the queries or the ability to trace the information flow is required. The detection of CTH in [18] is based on the source code of an application. Consequently, a new approach is needed to discover instances of the CTH in a database log. However, this is not trivial. As Example 1 has shown, without having the results of the first query one does not see dependencies of a sequence of SQL statements with certainty. The solving solution for CTH in [19] depends on the stage of the software development when the antipattern is discovered. If it is found early in the development, the authors suggest re-organizing the database schema. For distributed systems where one cannot do this, it is possible to reduce the number of remote database calls by using the Adapter pattern [20]. For designs with large intermediate results, an alternative is to create a new association that leads directly to the final result.

These solutions depend on a concrete case and are not automatic. Furthermore, they prevent future CTH occurrences, but do not solve CTH occurrences in a log.

4 PATTERNS AND ANTIPATTERNS

In our context, it is indispensable to have a rigid definition of 'pattern', to build a detection method. The notion of pattern is important in the context of antipattern detection, because antipatterns are patterns with negative effects. This section also introduces certain properties of patterns. We also need precise definitions of the Stifle and CTH antipatterns, to facilitate their detection and automatic solving. This section contains these definitions.

4.1 Database Pattern

As pointed out in Example 1, we see a pattern as a sequence of queries which represents certain functionality. Starting with the definition of patterns from software engineering where this function is a pattern, we first describe properties of patterns in databases informally. This discussion then leads to a formal definition of pattern that is then used throughout this article.

4.1.1 Intuition

In software engineering, a pattern is a recurring solution schema to a standard problem deployed in a certain context [4]. In this paper, a pattern is a sequence of SQL queries in a query log.

Example 7. Think of the database of a shoe retailer. Buying a pair of shoes results in the following sequence of steps that require interaction with the database:

1. Scan a barcode of the shoes.
2. Given the barcode, find the size and the model.
3. Write the purchase into the Sales table.
4. Decrease the count of the pairs currently available.

As shown below, Steps 2, 3 and 4 result in different queries forming one pattern. As these steps refer to the same business process, a common implementation is to have a procedure bundling the steps in one transaction:

```
CREATE PROCEDURE BUY
(BARCODE IN NUMBER) AS BEGIN
  SELECT MODEL, SIZE into curr_model, curr_size
  FROM BarCodesInfo  WHERE ID = BARCODE;
  INSERT INTO SALES (datetime, barcode, seller)
  VALUES (curr_time, BARCODE, curr_user);
  UPDATE InPresence  SET count = count - 1
  WHERE model = curr_model and size = curr_size;
END BUY;
```

Now every sale will cause these three SQL requests occurring consecutively. The only difference between occurrences is the parameter values, like the barcode. Summing up, to be an instance of a pattern, a sequence of SQL requests should

1. come one after another (in the log file).
2. have short time between them.
3. have a rather frequent occurrence.
4. be from the same user.

Regarding the last item, if the log does not contain information on the users, we assume that one user has issued all queries. A consequence could be that we will deem queries indeed coming from different users a pattern. However, we hypothesize that this phenomenon will be infrequent – it would mean that different users issue roughly the same queries at about the same time. Our case study will address this issue.

4.1.2 Similarity of SQL Queries

Given a query log, patterns are not known in advance, but need to be discovered first. So, it is necessary to find similar sequences of queries within the log. We define similarity of a sequence of queries as follows:

Definition 1. Two sequences of SQL statements are similar if they contain similar queries in the same order.

We now define query similarity. Since a query is a request written in SQL, it seems promising to use methods for the discovery of similar pieces of source code to identify similar queries [21]. Respective approaches from software engineering like Code Clone Detection [22] rely on syntax trees [23]. As Example 7 has shown, while the structure of the queries remains the same (i.e., the inner nodes of the syntax tree are identical) the parameters are most likely different. Thus, one should not consider these values when computing query similarity. We rely on the notion of skeleton tree (or skeleton query [9], SQ). It is obtained from a syntax tree by replacing all parameters in the leaf nodes with placeholders.

Definition 2. SFC ("Skeleton From-Clause"), SWC and SSC are skeletons of FROM, WHERE and SELECT clauses in the corresponding SQ.

Definition 3. SC, FC and WC are the SELECT-, the FROM- and the WHERE-clause of the query.

This differentiation will let us introduce a more sophisticated classification of patterns and antipatterns later.

Definition 4. A query template is a triple consisting of skeleton subtrees (SFC, SWC, SSC).

Definition 5. Two skeletons SQ_1 and SQ_2 are equal to each other iff
$(SFC_1 = SFC_2) \wedge (SWC_1 = SWC_2) \wedge (SSC_1 = SSC_2)$

In the following, we need a notion of similarity of queries/their skeletons. At first sight, small deviations in

structure seem tolerable. However, a pattern tends to be created by one software application, i.e., the queries have the same structure. In consequence, we start by focusing on the case that query skeletons are equal, see Definition 6. Our case study will examine whether this definition is adequate or should be refined.

Definition 6. Two queries are similar iff their skeletons are equal.

Example 8. Consider the following queries:

```
SELECT a, b FROM T WHERE a = 0 AND b >= 3
SELECT a, b FROM T WHERE a = 10 AND b >= 5
```

A representation of the SQ for both queries then is:

```
SELECT a, b FROM T
WHERE a = <num> AND b >= <num>
```

4.1.3 Definition and Properties of a Pattern

We now define the notion of pattern in our context.

Definition 7. A pattern is a sequence of query templates $(SQ_1, \dots, SQ_n)$. Thus, a pattern is a sequence of query skeletons $(SQ_1, \dots, SQ_n) = ((SFC_1, SWC_1, SSC_1), \dots, (SFC_n, SWC_n, SSC_n))$

Definition 8. An instance $(Q_1, \dots, Q_n)$ of a pattern is a sequence of queries in the query log such that

- $user(Q_1) = user(Q_2) = \cdots = user(Q_n)$
- $time(Q_1) \leq time(Q_2) \leq \cdots \leq time(Q_n)$
- $\forall i, 1 \leq i \leq n, \nexists Q_x \notin (Q_1, \dots, Q_n)$ *where* $user(Q_x) = user(Q_i) \wedge time(Q_i) \leq time(Q_x) \leq time(Q_{i+1})$

The last axiom states that there are no other requests from the same user within time window $[time(Q_1); time(Q_n)]$.

Definition 9. The frequency of a pattern in a log is the number of its instances occurring in the log.

Definition 10. The userPopularity of a pattern in a log is the number of users who have submitted queries being instances of the pattern.

Frequent patterns with low *userPopularity* are an important phenomenon. For instance, one might perceive such patterns as bias when identifying hot spots of user interests. One hypothesis that explains the occurrence of such a pattern is that a database is copied piece by piece. In our case study, we will examine how often such patterns occur and discuss the phenomenon further.

4.2 Definitions for Antipatterns

We now give a formal definition of the selected antipattern types. In general, an antipattern is a pattern which introduces negative consequences. Therefore, antipatterns have all the properties described in Section 4.1.3. For each selected antipattern, we provide a detection rule. If an antipattern has a cleaning solution, we consider it *solvable*.

4.2.1 The Stifle Antipattern

Our literature review (see Section 3.1.1) has yielded the following specific characteristics of the Stifle:

1. Small average number of result rows or (in case of update statements) of rows affected,
2. High number of similar database queries.

The nature of the Stifle is that all its queries refer to one object. Each query has few result rows, typically tuples with a foreign-key relationship with this object, and the queries cause repeated similar requests. Thus, applications create Stifle instances most likely using databases in an object-oriented fashion similarly to the `get()` or `set()` method. These methods refer to specific objects, i.e., to rows in a database table identified by the same id. Thus, we presume that the Stifle consists of one equality predicate which filters data using an attribute which is a key.

Definition 11. A Stifle antipattern is a pattern $(SQ_1, \dots, SQ_n)$ such that

- $CP_1 = CP_2 = \cdots = CP_n = 1$
- $\theta_1 = \theta_2 = \cdots = \theta_n = 'equality'$
- $filCol_1, \dots, filCol_n$ are key attributes.

CP is a count of predicates, θ is the comparison operator in the predicate, and *filCol* is the filter column in the predicate. Note that Definition 11 relies on a database schema, to distinguish between key and non-key attributes. We could have omitted the third axiom in principle: This would have simplified things, but with the potential drawback of some false positives. Our solving scheme for this antipattern depends on its form. We differentiate between classes of the Stifle, based on the clause where the queries differ. Such a difference may be either in the WHERE, the FROM, or the SELECT clause. We now describe them, followed by our solution to clean the log.

4.2.1.1 DW-Stifle

The first case is that the statements in an instance of a pattern have equal SELECT and FROM clauses, but a different WHERE clause. We refer to this as DW-Stifle ('different WHERE' Stifle).

Example 9. The following is a DW-Stifle antipattern:

```
SELECT name FROM Employee WHERE empId = 8;
SELECT name FROM Employee WHERE empId = 1;
```

Hence, DW-Stifle is a pattern with the same SC, FC and SWC but different values in the WHERE clause. The formal definition is as follows:

Definition 12. A DW-Stifle is a Stifle $(SQ_1, \dots, SQ_n)$ such that

- $SC_1 = SC_2 = \cdots = SC_n$
- $FC_1 = FC_2 = \cdots = FC_n$
- $SWC_1 = SWC_2 = \cdots = SWC_n$
- $WC_1 \neq WC_2 \dots \neq WC_n$

Our cleaning solution is to compose one query with all filtering conditions in the WHERE clause.

Example 10. The cleaning solution for Example 9 is :

```
SELECT empId, name FROM Employee WHERE empId IN (8, 1);
```

Compared to the solving solution in Example 6 we now get one SQL statement instead of several ones.

4.2.1.2 DS-Stifle

If an instance of the Stifle has a sequence of queries with equal FROM and WHERE clause, it is a DS-Stifle ('different SELECT' Stifle).

Example 11. A DS-Stifle instance is as follows:

```
SELECT name FROM Employee WHERE empId=8;
SELECT address, phone FROM Employee WHERE empId=8;
```

Definition 13. A DS-Stifle is a Stifle $(SQ_1, \dots, SQ_n)$ with the following characteristics:

- $SSC_1 \neq SSC_2 \neq \cdots \neq SSC_n$
- $FC_1 = FC_2 = \cdots = FC_n$
- $WC_1 = WC_2 = \cdots = WC_n$

To solve this, we union the SELECT clauses, as follows.

Example 12. The cleaning solution for Example 11 is :

```
SELECT name, address, phoneNumber
FROM Employee WHERE empId = 8;
```

4.2.1.3 DF-Stifle

Patterns with different FROM statements are named DF-Stifle ('different FROM' Stifle). The formal definition is as follows:

Definition 14. A DF-Stifle is a Stifle $(SQ_1, \dots, SQ_n)$ where

- $SFC_1 \neq SFC_2 \neq \cdots \neq SFC_n$
- $FC_1 \neq FC_2 \neq \cdots \neq FC_n$
- $WC_1 = WC_2 = \cdots = WC_n$

Inequality in the FROM clause could mean redundant database design as we point out in Example 13. Thus, Example 14 illustrates our solving scheme:

Example 13. The following queries select information on the same real-world object from different tables:

```
SELECT name FROM Employee WHERE empId = 8;
SELECT address FROM EmployeeInfo WHERE empId = 8;
```

Example 14.

```
SELECT E.name, EI.address
FROM Employee as E INNER JOIN EmployeeInfo as EI
ON E.empId = EI.empId WHERE empId = 8;
```

At first sight, a large number of DS-Stifle instances suggest a refactoring of the underlying tables or introducing one or several views.

Example 15. To cope with the situation illustrated in Example 14, the following view might seem helpful:

```
CREATE VIEW EmployeeView AS
SELECT E.name, EI.address FROM Employee as E
INNER JOIN EmployeeInfo as EI ON E.empId = EI.empId
```

Now one can access the view EmployeeView instead of the underlying tables. However, this suggestion does not address our specific problem, namely solving instances of the antipattern in the log a posteriori. We use the method as in Example 14 to solve instances of the DF-Stifle.

4.2.2 The Circuitous Treasure Hunt Antipattern

The distinctive feature of the CTH antipattern is dependency of the queries. As re-querying is not feasible (see Section 1) to identify dependencies in sequences of SQL queries, we need a different approach to detect CTHs. The sequence and structure of the individual queries of a CTH contain hints that allow detecting CTH candidates. In a CTH, the result of the first query is an input parameter for a subsequent one. Hence, we require that

- there are attributes in the SELECT clause of the first query used in the WHERE clause of the other query,
- the second query features an equality predicate in the WHERE clause, to refer to the tuple exclusively.

However, relying solely on these conditions could yield false positives. Without re-querying one can only detect candidates. So the following is a definition of 'CTH candidate', not of (real) CTH.

Definition 15. A CTH candidate is a pattern $(SQ_1, \dots, SQ_n)$ such that:

- $SQ_1 \neq SQ_2$
- $CP_2 = \cdots = CP_n = 1$
- $\theta_1 = \dots = \theta_n = 'equality'$

Our respective detection method looks for patterns which satisfy Definition 15. The decision whether a candidate is a *real CTH* requires domain knowledge. Our case study will quantify the share of false positive CTHs which our heuristics produces.

5 IMPLEMENTATION

In this section, we describe the realization of our approach. We first give an overview of the architecture of our framework that cleans antipatterns. We then introduce the components of the processing pipeline in more detail. There is a web page of our framework[1] where one can find its documentation, a test set and the source code.

The purpose of our framework is to analyze query logs. Depending on the analysis target, we intend to find *query templates* or patterns (series of *query templates*) within the log, or identify and solve antipatterns. Fig. 1 shows the respective workflow. Rectangular boxes stand for input data, rounded boxes for processing steps, and gray boxes for results. There are several results extracted from a SQL log. In contrast, the log is the only input. This current section is supposed to give the reader an impression of the potential of our solution as well as an impression of how to use it. It also describes parameters of our framework. We now discuss some relevant details.

5.1 Original Query Log

The original query log is the *only* input that is required. Our approach does not need to have access to the database and does not introduce any load overhead. We require the log to consist of SQL statements together with their execution time. However, the more additional information is available, the better one can perform the analysis. For example, if we can differentiate between users who have issued queries, we can obtain the user popularity of a pattern (see the discussion in 4.1.3). Nevertheless, our framework is operational without this information as well.

5.2 Deleting Duplicates

The first processing step is deleting duplicate queries. We perceive duplicates as unintended errors. Consequently, we record the number of duplicate removals in the result statistics. This is because a large number of them may indicate a refactoring of a particular applica-

[1] https://dbis.ipd.kit.edu/2500.php

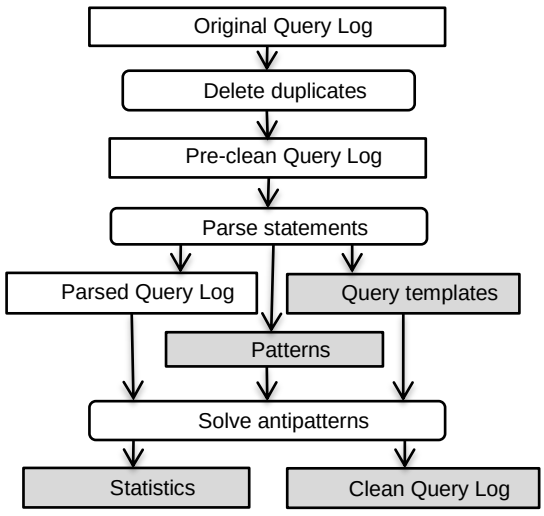

Fig. 1. Processing Steps

tion.

We define *duplicates* as identical statements with a small difference in time. From a conceptual point of view, we argue that two identical statements executed by the same user only stand for the same information need in case the time difference is smaller than the threshold. We propose setting the threshold to the minimum value, allowing to find most of the duplicates. Section 6.2 will discuss respective empirical results.

5.3 Parsing Statements and Parsed Query Log

After deduplication, there may still be syntactically incorrect query statements. In this step, we parse all queries and build a syntax tree for each of them. If the parsing finds syntax errors, the process will not consider the statement any further. We also exclude non-select statements. The parsing procedure also extracts the SELECT, FROM and WHERE subtrees and their skeleton forms for each statement. Table 2 contains an example of a parsed log (without skeletons of FROM, WHERE and SELECT). Each row of a parsed log also contains the link to a pattern (from the Patterns block) and a query template (from the Query templates) a statement belongs to. If a parsed statement satisfies the definition of an antipattern (Definition 11 to Definition 15), it is marked as an antipattern of the respective type.

5.4 Query Templates and Patterns

We compute statistics for each template and pattern using the *frequency* and *userPopularity* properties (Definition 9 and Definition 10, see Section 4.1.3). In addition to these attributes, a pattern has a property indicating whether it is an antipattern and, if so, its type. As the next step, instances of the Stifle need to be solved.

Our framework can be extended to accommodate other antipatterns. In the presence of a new antipattern, one first comes up with its formal definition, often after a literature review. Based on the definition, one provides a detection rule and, if possible, a solving solution. For instance, suppose that we want to extend our framework with the "Searching nullable columns" (SNC) antipattern [24]. An example of it is as follows:

```
SELECT * FROM Bugs WHERE assigned_to = NULL
SELECT * FROM Bugs WHERE assigned_to <> NULL
```

Since neither equality nor inequality return true when comparing a value to a null value, one needs another operation when searching for a null value, IS NULL or IS NOT NULL. Hence, if this is the intention, the previous statements should be rewritten as follows:

```
SELECT * FROM Bugs WHERE assigned_to IS NULL
SELECT * FROM Bugs WHERE assigned_to IS NOT NULL
```

We now provide a formal definition of SNC.

Definition 16. A SNC is a pattern (SQ_1) where

- WC_1 *consists of* "$NULL$"
- $\theta_1 =' equality' \vee 'inequality'$

The solving solution is rather straitforward: replace "$= NULL$" with "$IS\ NULL$" and "$<> NULL$" or '"$!= NULL$" with "$IS\ NOT\ NULL$".

Now one needs to include the new detection rule based on Definition 16 in the parse step, as we have done with antipatterns described in Section 4.2. Since there is a solving solution as well, one can include it in the step "Solve antipatterns". From now on, each SNC detected will be solved.

5.5 Solving Antipatterns, Clean Query Log and Statistics

The approach iterates over the whole log, and for every pattern detected, it checks whether it is an antipattern. If so, it solves it, following the solving solutions described earlier (e.g., Example 10 or Example 12). The procedure returns a cleaned query log and statistical information regarding antipatterns solved: how many of them have been encountered in the query log, how many have been solved. The following is an example of this procedure.

TABLE 2
A parsed query log

#	Statements	Type
1	`SELECT E.Id FROM Employees E WHERE E.department = 'sales'`	CTH
2	`SELECT E.name, E.surname FROM Employees E WHERE E.id = 12`	CTH, DW-Stifle
3	`SELECT E.name, E.surname FROM Employees E WHERE E.id = 15`	CTH, DW-Stifle
4	`SELECT E.name, E.surname FROM Employees E WHERE E.id = 16`	CTH, DW-Stifle

TABLE 3
A clean query log

#	Statements	Type
1	`SELECT E.Id FROM Employees E WHERE E.department = 'sales'`	CTH
2	`SELECT E.name, E.surname FROM Employees E WHERE E.id IN (12, 15,16)`	CTH

Example 16. Table 2 contains a query log after parsing. Queries 2, 3 and 4 form a DW-Stifle antipattern, which is solvable. The first four queries are CTH candidate.

In the next step, the instance of DW-Stifle is rewritten as one request, as shown in Table 3.

In the example, Queries 2, 3 and 4 belong to both, DW-Stifle and CTH. However, since we do not provide a solving solution for CTH, there is no conflict regarding what to solve. If a certain subset of queries shows multiple *solvable* antipatterns, we perform our solving procedure in the order of queries occurring in the log. Put differently, solving starts with the antipattern which appears in the log first.

After one cleaning step, there can be further solvable antipatterns. To check this, one needs to parse statements again and possibly solve the antipatterns. In our case, the experiments have not indicated any necessity to do so: After the first cleaning, the number of solvable antipatterns contained in the log has been 0.09 %, which is negligible.

6 A CASE STUDY WITH SKYSERVER

This section reveals insights on the existence and frequency of antipatterns in a real-world query log. The particular objectives of our case study are:

1. Answer the question how many patterns or antipatterns are contained in a large real-world log;
2. Give meaning to the most popular patterns;
3. Hypothesize on the rationale behind patterns based on their frequency and user popularity;
4. Determine the positive detection rate of the CTHs;
5. Study the importance of metadata (users IPs and sessions) on pattern and antipattern detection.
6. Showcase the influence of cleaning antipatterns on subsequent analyses.

6.1 Appropriateness of the SkyServer Log for a Case Study

We have used the SkyServer query log for our case study, since it is a large scientific data set available to the public, and, to our knowledge, it is the only one with this characteristic. The log provides extensive information regarding all requests. Besides the actual SQL statement and its timestamp, it contains the user IP, a label of the user session and the number of result rows. See http://skyserver.sdss.org/log/en/traffic/sql.asp for a description of the SQL log columns. The public availability of this log allows for easy verification or extension of our results by the scientific community. We analyze the SkyServer log of SQL statements for five years from 2003 to 2008. It consists of 42 million queries from about 47 thousand users.

6.2 Choosing the Duplicate Time Threshold

For our analysis, we need to choose a time threshold for duplicate queries. We set this threshold by testing several values with a sample data set of 10^5 queries (cf. Table 4). Most duplicates are already identified when using 1 second as the threshold. The difference between the effect of this value and setting it to infinity is about 0.5%, which we deem insignificant. This means that duplicates indeed are requests not intended by the user, as we have hypothesized in Section 5.2. Most duplicate queries are submitted to the database within a second. We conclude that they result from web-form reloads or from errors in applications.

TABLE 4
Experiments with threshold parameter for deleting duplicates

threshold	log size	% of original size
Original Log	5,748,440	100
1 sec	5,515,737	95.95
2 sec	5,515,737	95.95
5 sec	5,512,468	95.89
10 sec	5,507,233	95.80
Non restricted	5,484,746	95.41

When increasing the duplicate time threshold, the more identical queries from one user are classified as duplicates, and the slower the procedure that removes duplicates. Setting the threshold to infinity is not always good, since two identical queries with a big time difference between them might not be a duplicate after all, but reflect user intention. Since user behavior may differ between databases, each SQL log analysis may require its own threshold value. Tests similar to the one just described should allow to determine this value.

6.3 General Results

From the 42 million queries of the raw log, we extract 40 million, which are not DML or DDL, and which do not contain syntax error. After deleting duplicates, the log contains 38.5 million queries (see Table 5). Our analysis of the log indicates that our definition of query similarity (Definition 5) is adequate: manual investigation of the results has revealed that any query templates which are structurally different from each other also feature different requests. Overall, cleaning a log with our approach has resulted in 27.5% size reduction. This is significant. The number indicates that there is a large share of antipatterns, and that our antipattern definitions are valid. In more detail, we have uncovered 1018 distinct DW-Stifles, 656 DS-Stifles and 487 DF-Stifles. Among 50 candidates for CTH, 28 turn out to be real ones (see Section 6.9). The instances of the antipatterns cover about 7.5 million statements.

To complete this current series of experiments, we have conducted one experiment on how rewriting antipatterns influences the runtime of the queries. To do so, we picked 10222 queries which form solvable antipatterns (Stifles). After rewriting, only 254 queries remain a reduction by a factor of 40. Running the original 10222 queries takes 4450 seconds, while the rewritten 254 queries require 152 seconds, 29.27 times faster. This effect is due to reduced network overhead and database resources.

Another benefit due to this size reduction is a decreased load for subsequent downstream analyses, see Section 6.9.

TABLE 5
Results overview

Property	Value
Size of original query log	41,998,253
Count of Select queries	40,177,133 (95.9 %)
Size of log after deleting duplicates	38,529,871 (91.74%)
Final log size	30,454,778 (72.51%)
Count of patterns	176,110
Maximal pattern frequency	3,349,709
Count of distinct DW-Stifle	1,018
Count of queries in all DW-Stifle	6,326,863
Count of distinct DS-Stifle	6,562
Count of queries in all DS-Stifle	1,281,936
Count of distinct DF-Stifle	487
Count of queries in all DF-Stifle	212,103
Count of distinct candidate CTH	50
Count of queries in all candidate CTH	424,792
Count of distinct real CTH	28
Count of queries in real CTH	435,251

6.4 Effects of SQL Log Cleaning

To evaluate the effectiveness of cleaning antipatterns, we compare the most popular patterns before and after running cleaning procedure. Fig. 2 (a) tells us that there are 9 antipatterns among the 30 most popular patterns. If we consider only the top-15 patterns, we even find six of them to be antipatterns. Table 6 shows the most frequent ones. The *frequency* of discovered antipatterns highlights the importance of cleaning the log. The *solvable* antipatterns (DS-Stifle, DF-Stifle and DW-Stifle) cover about 19.2% of the query log, a significant value. Furthermore, discovering antipatterns in a query log in this case allows detecting users who perform requests with such antipatterns. Operators of the database could then contact these individuals and inform/train them accordingly.

As Table 6 has revealed, the most frequent antipattern is DW-Stifle. All antipatterns filter the table photoPrimary by the internal attribute objId, which is not a notion from astronomy. Hence, we hypothesize that the antipatterns cannot be interpreted as user intentions, while patterns which are not antipatterns can. We will discuss this assumption further in the next section. We observe that those antipatterns do not have high user popularity – most of them come from a few distinct IP addresses. We conclude that the software generating the antipatterns is proprietary and not part of the SkyServer infrastructure.

6.5 Interpretation of Patterns

In this section, we discuss the meaning of the most popular patterns. We demonstrate that, unlike antipatterns, patterns represent user interests. This is an indication that we have curbed the extent of bias introduced by antipatterns significantly.

Table 7 contains the most popular patterns in the query log after removing the antipatterns. All five patterns perform spatial search, i.e., look for objects in some part of the sky. These queries are meaningful for domain experts. In other words, pattern extraction reveals particular ways users use the database. We find it remarkable that the most popular patterns come from very few users. None of the patterns created by the SkyServer Web interface does fall in the top 5. Such patterns occur at rank 12 and 17. Rank is a position in a list of patterns sorted by frequency. The most frequent pattern has rank 1; the next one in popularity has rank 2, etc.

According to Fig. 2 (b), our study reveals a large number of frequently occurring patterns with low user popularity. In particular, 23 out of the 40 most popular patterns were run only by one user. The instances of these patterns perform a sliding window search, i.e., consecutive requests for certain objects with disjoint filtering conditions. From now on, we refer to this as sliding window search pattern (*SWS pattern*). We do not classify the SWS pattern as an antipattern since it does not have a negative performance effect. Our explanation why this pattern occurs is that, due to SkyServer database restrictions, users' access data piece-wise, downloading a significant part of the database.

Clearly, SWS detection depends on $frequency$ and $userPopularity$ thresholds. We now briefly discuss the effect of these parameters. In a nutshell, they reflect how rigid one wants to be in SWS cleaning. If we set 'frequency' higher and 'userPopularity' lower, we will get rid only of the most obvious SWS. Only patterns which are frequent and are due to, say, one or two users will be filtered out. Decreasing 'frequency' and increasing 'userPopularity' means more major cleaning. This is because patterns of medium frequency which come from more users will be labeled as SWS. Table 8 contains the numbers for our case study. The $frequency$ threshold is in relative terms (%). A cell of a table indicates how much of the log we classify as SWS with the respective $frequency$ and $userPopularity$

TABLE 6
The most popular antipatterns

#	Frequency	Type	First skeleton statement	Second skeleton statement	Distinct IPs
1	1,454,207	DW	`SELECT rowc_g, colc_g FROM photoprimary WHERE objid=num`	`SELECT rowc_g, colc_g FROM photoprimary WHERE objid=num`	2
2	1,410,696	DW	`SELECT rowc_r, colc_r FROM photoprimary WHERE objid=num`	`SELECT rowc_r, colc_r FROM photoprimary WHERE objid=num`	3
3	1,044,958	DW	`SELECT rowc_i, colc_i FROM photoprimary WHERE objid=num`	`SELECT rowc_i, colc_i FROM photoprimary WHERE objid=num`	1
4	559,450	DS	`SELECT rowc_r, colc_r FROM photoprimary WHERE objid=num`	`SELECT rowc_g, colc_g FROM photoprimary WHERE objid=num`	2
5	558,930	DS	`SELECT rowc_g, colc_g FROM photoprimary WHERE objid=num`	`SELECT rowc_r, colc_r FROM photoprimary WHERE objid=num`	2

TABLE 7
The most popular patterns

#	Frequency	Coverage (%)	Skeleton statements	Description	Distinct IPs
1	3,349,709	8.69	`SELECT g.objid, … FROM photoobjall as g JOIN fgetnearbyobjeq(@ra, @dec, @r) as gn on g.objid=gn.objid left outer join specobj s on s.bestobjid=gn.objid`	Gets objects within @r arcmins of an Equatorial point (@ra,@dec)	1
2	3,082,742	8.0	`SELECT p.objid, … FROM fgetobjfromrect(@ra1, @dec1, @ra2, @dec2) n, photoprimary p WHERE n.objid=p.objid and r between num and num`	Gets objects from rectangle area with radius between two values.	19
3	2,179,250	5.65	`SELECT count(*) FROM photoprimary WHERE htmid>=@htm1 and htmid<=@htm2`	Gets the count of objects within a range of spherical triangles (special search)	1
4	2,099,560	5.44	`SELECT p.objId, … FROM fgetnearbyobjeq((@ra, @dec, @r) n, photoprimary p WHERE n.objid=p.objid`	Get information about the objects within @r arcmins of an Equatorial point (@ra,@dec)	1
5	674,071	1.75	`SELECT ra, … FROM fgetnearbyobjeq((@ra, @dec, @r) n, photoprimary p WHERE n.objid=p.objid`	Get information about the objects within one fraction of a scan strip observed at one time (limited by observing conditions). It is also some sort of special search	1

thresholds. The numbers are in line with our explanation.

The discovery of SWS patterns is important for user-interest finding. Queries within such a pattern do not overlap in the area of the data space accessed. However, instances of these patterns produce a specific uniform noise, which one can exclude in subsequent analyses. An alternative to exclusion is a union of the filtering conditions, i.e., replacing all these queries with one that yields the same result.

We think that SWS patterns also bog down prediction quality of association-rule mining. Suppose that we want to suggest the next query based on the previous one. If the learning set contains SWS pattern queries, a query recommendation system would suggest a query with a disjoint filtering condition. However, this kind of behavior (sliding window search) is a "machine download", which does not require query-recommendation assistance. Humans on the other hand would benefit from query recommendation. [9] devotes attention to this issue. However, their recommendations only consider the duration of user sessions, not the shape of queries. An extension taking SWS patterns into account could distinguish humans and "bots" with more accuracy. In our future work, we will study the influence of cleaning the learning set on the quality of query recommendation.

6.6 CTH Detection

Having discussed the automatic detection of the Stifle, we now turn to CTH antipatterns. With our solution, we discover 50 candidate antipatterns. As mentioned, the decision whether such a pattern is a CTH requires domain knowledge. The number however is small, at least in this current case, making respective intellectual effort tolerable. Thus, we analyze those few patterns by hand

TABLE 8
SWS coverage depending on frequency and user popularity thresholds

UserPopularity \ Frequency	10%	1%	0.1%	0.01%
1	8.7%	18.7%	31.2%	35.4%
2	8.7%	18.7%	36.0%	40.9%
4	8.7%	18.7%	40.3%	45.6%
8	8.7%	18.7%	40.7%	46.1%
16	8.7%	18.7%	41.0%	46.3%

and conclude that 28 out of 50 are real CTH antipatterns. We deem a CTH candidate a real antipattern if the decision regarding the next statement is predefined. The following example is an illustration.

Example 17. Consider an instance of two CTH candidates:

TABLE 9
CTH candidate 1

#	Statements	Time
1	`SELECT name, type FROM DBObjects WHERE type='U' AND name NOT IN ('LoadEvents', 'QueryResults') ORDER BY name;`	13.06.07 12.18.46 PM
2	`SELECT description FROM DBObjects WHERE name='Galaxy'`	13.06.07 12.19.13 PM

The instances of CTH Candidate 1 comply with SkyServer Web interface functionality: First, a user looks for all tables in a database, and then choses table 'Galaxy' and query it. The time difference between two queries indicates that the second query has not been issued before

the first one – the user apparently has reflected for a while which table he/she is interested in.

TABLE 10
CTH candidate 2

#	Statements	Time
1	`SELECT * FROM dbo.fGetNearestObjEq (145.38708,0.12532,0.1);`	18.09.07 11.25.00 AM
2	`SELECT plate, fiberID, mjd, SpecObjID FROM SpecObjAll WHERE SpecObjID =75094094447116288`	18.09.07 11.25.00AM

The queries in CTH Candidate 2 in contrast run directly one after another, with no time difference. The first query performs returns the closest object for a certain point; the second query then instantly asks for the object the first query has returned. Even if the count of the second queries is not equal to the number of rows the first query has returned, this can only mean that there is some logic deciding which objects from the first result are fetched. This logic relies on the first result, so this indicates a dependency between the two queries. Thus, this is an instance of CTH.

We have distinguished between real CTHs and CTH candidates, due to our rigid interpretation. A more generous interpretation results in more true CTHs. While it seems reasonable to evaluate a CTH detection method objectively by measuring precision and recall, this is not feasible in our case. These metrics require a ground truth, i.e., false positive and false negative CTHs must be known. To get there, one would have to interview thousands of SkyServer users on their exact intentions, make sure that their answers are clear (even though many of them might not be trained well enough to this end) etc. Thus, all we can claim based on our study is that our detection method can identify CTHs within a query log.

Fig. 2 (d) shows false positives and real CTHs, depending on their frequency and user popularity. In this visualization, we observe a dependency between user popularity and the property of being a CTH. However, this is not an indicator for real CTH for sure: Widely used software could introduce instances of the CTH as well.

6.7 Feedback from Domain Experts

In order to assess the usefulness of our results, we have conducted an experiment with domain experts. We have provided the list of the most popular patterns and antipatterns and have asked the experts to explain their meaning. They did not have any information from our side regarding whether we consider a pattern an antipattern. We also had not explained to them what antipatterns are. The experts have stated that all patterns (not antipatterns) are meaningful from their point of view. They also deem antipatterns follow-up queries, where a user has first obtained objIDs (see Table 6) with a previous query and asks for more data. This is exactly what CTH does, and this has been our hypothesis as well. The fact that, based on our results, the domain experts have come to the same conclusion independently proves that our framework is indeed able to find antipatterns in a real-world query log.

6.8 Reduced Information in the Log

Not every SQL log contains all the information that is part of the SkyServer log. Therefore, our framework only requires SQL statements with timestamps as input. In this section, we study the result quality with this minimum input. More specifically, we compare the results with two input data sets:

1. SkyServer query log with user-session information
2. SkyServer query log containing only SQL requests and timestamps.

We expect insignificant changes in pattern frequency, for the following reason. The statements belonging to one occurrence of a pattern have a very small time difference and therefore come one after another in time order anyway. Fig. 2 (c) shows the most popular patterns and antipatterns with those two different inputs. All in all, the patterns keep their frequency of occurrence without user-session information. It means that, with high probability, requests from one user with a small time difference would come one after another. So, results seem to be meaningful for a query log which standard tools can collect. In terms of log size after cleaning, the difference is insignificant as well: The count of queries in the result set for the experiment with minimal input data is 0.36% less than for the original.

However, without information regarding users, our approach bears certain limitations. For instance, it is impossible to find patterns which have low user popularity and perform sliding window search. This means that we cannot reliably detect and remove this specific noise.

To conclude, we now discuss the options if there are no timestamps in the query log. Our processing requires timestamps only for duplicate detection. Otherwise, exact timestamps are not necessary. On the other hand, we still need information according to which we can order the requests by time. (A pattern is a sequence of statements, not a set.) Statements within antipatterns must be ordered; this is essential for their identification.

6.9 Effects on Downstream Analysis

Even though it is not the core topic of this article, we now present some insights into the influence of the cleaning on subsequent analyses. To this end, we extract 1.3 million queries from the log and reproduce the experiment described in [1]. It detects user interests based on the log. They cluster queries, using the overlap of the data space accessed by two queries as their distance measure. More specifically, the bigger the overlap, which ranges from 0 to 1, the smaller is the distance. Queries with a distance smaller than a threshold go to the same cluster. We run the experiments with different threshold values from 0.1 to 0.9 with a step of 0.1, using three variants of that sample:

1. Raw query log (1.3 million queries)

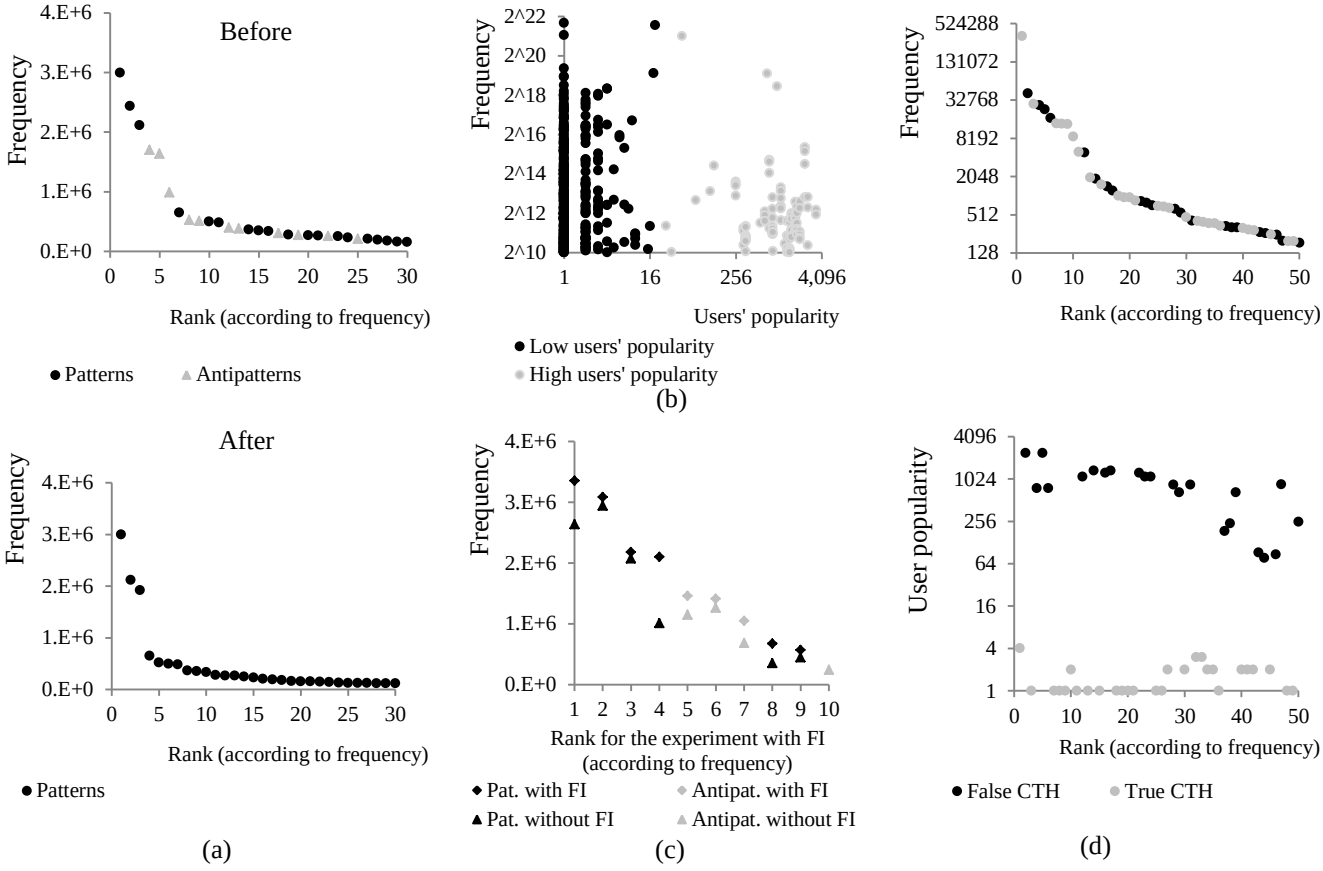

Fig. 2. (a) The most popular patterns before and after cleaning the log; (b) Frequency and user popularity of the patterns; (c) The experiments with and without user session information; (d) Possible and real CTH antipatterns

2. Removal query log, obtained from the raw query log by removing antipatterns (0.89 million queries).
3. Clean query log, obtained from the raw query log by cleaning antipatterns (1 million queries).

In the case of cleaning, we do not delete antipatterns from a log, but rewrite them, as discussed in Section 4.2. Hence, the removal log is smaller than the clean one.

Fig. 3 shows the results. Varying the threshold value (from 0.1 to 0.9) has little impact on the number of clusters. This is because the distance metric which calculates the overlap of two queries very often yields 0 (queries are identical) and 1 (queries do not have any overlap). The number of occurrences of other distance values has been very low in our experiments. The clusters in the raw log are too numerous to be analyzed individually. For example, for threshold value 0.9 there are 1393 of them. Most of them also are relatively small. The log without antipatterns ("removal") yields bigger and at the same time fewer clusters. This happens for the following reason: When removing antipatterns, we filter out a lot of small clusters formed by them. Hence, for the removal log we have got a number of clusters which one can analyze manually and interpret as user interests (51 clusters for threshold value

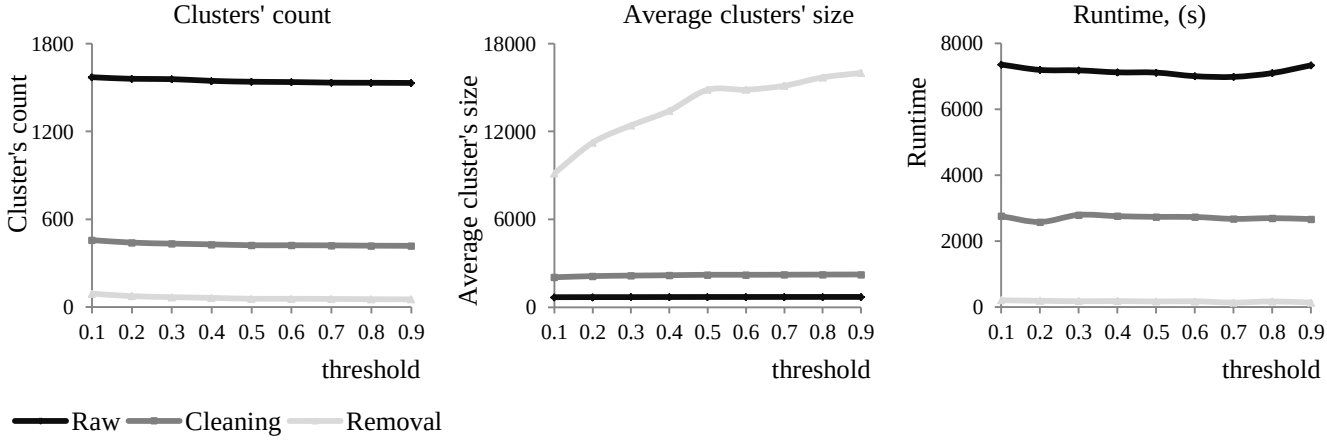

Fig. 3. Results of an experiment on query clustering

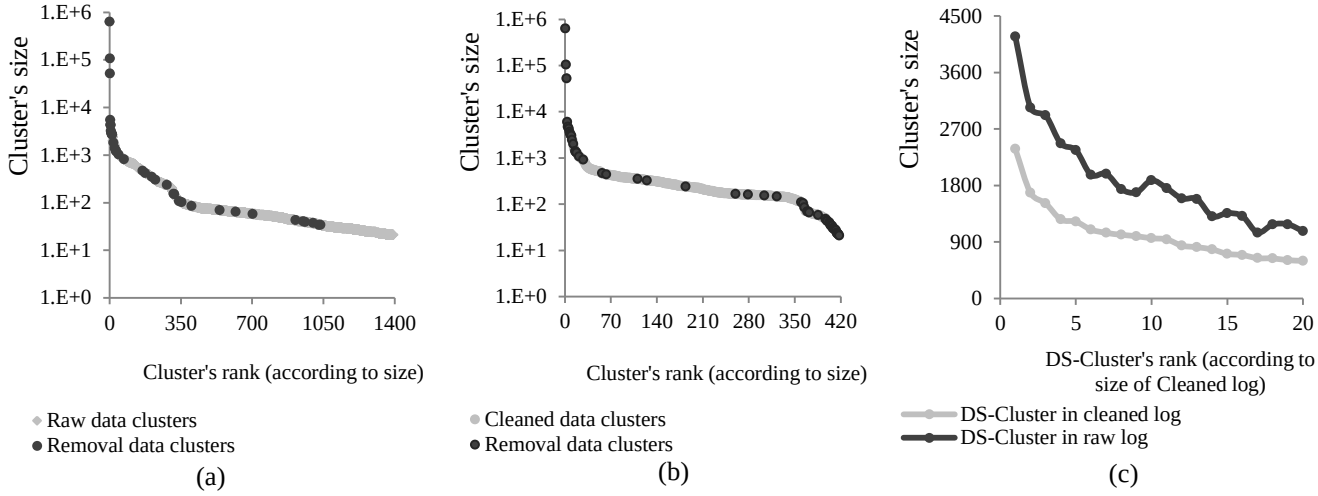

Fig. 4. (a, b) Clusters' sizes for experiments with raw, clean and removal data; (c) DS-Clusters' sizes for cleaned and raw log. Threshold = 0.9

0.9). We have carried out such an analysis and conclude that most clusters do reflect an area of user interest, as they refer to certain locations in the sky. We have found all clusters from the removal log in the raw log as well as in the cleaned log, see the middle gray dots in Fig. 4 (a, b). This indicates that removing antipatterns indeed is a way to get rid of noise that does not hamper subsequent processing. From the performance point of view, cleaning or removing of antipatterns is significant as well: The runtime curve of Fig. 3 indicates that the smallest log ("removal") gives way to the best time. This time, however, does not change linearly with the number of points, because clustering requires comparing one object (query) to the other ones. The complexity of the procedure in the worst case is $O(n^2)$.

The experiments with the cleaned log show that clusters with DS-Stifle instances are smaller. This is line with our prediction, i.e., 0. Fig. 4 (c) graphs the sizes of the top 20 biggest DS-Clusters in the clean and in the raw log. Clusters in the raw log are approximately two times bigger. For instance, the biggest DS-Cluster in the raw log consists of statements like:

```
 1.  SELECT text FROM DBObjects
WHERE name='photoobjall';
 2.  SELECT description FROM DBObjects
WHERE name='photoobjall';
```

Most queries in the clean log in turn consist of statements like:

```
SELECT text, description FROM DBObjects
WHERE name='photoobjall';
```

We for our part conclude that both removal and cleaning improve the quality of the data, and that the process as a whole is meaningful.

6.10 Threats to Validity

Regarding threats to validity, our goal is studying validity of our framework. Recall that three variants of the query log are the input of the clustering procedure. The experiment with the raw data, the query log as is, is the baseline. The framework (pre-processing) is a treatment and can be seen as an independent variable. The dependent variable is the clustering result. Since we use the same clustering algorithm with the same parameters for each dataset (raw, clean, removal), it is only the treatment that explains the difference in the results. Hence, the experiment has high internal validity.

External validity is the degree to which the results of an experiment can be generalized. To discuss this aspect fully, one would need to perform our processing on another SQL query log. Unfortunately, to the best of our knowledge, there is only one query log publically available – Sky Server data, which we are using for our current study already. To overcome this and prove that we have not arrived at our results by accident, we can try different samples of the query log and cluster them. We have performed such experiments and conclude that, for all datasets samples, clustering raw data yields much more clusters, with small average size. Put differently, the baseline experiment always returns noisy results compared to experiments with a clean or a removal log. A further investigation of external validity – hunt for another case study – is part of future research.

7 CONCLUSIONS

Knowing how a big database is used is highly important for its owner. Analyzing the SQL log and finding patterns is one promising approach in order to reveal such information. Antipatterns however might falsify such analyses; discovering antipatterns in the log is beneficial for refactoring and post-processing. To our knowledge, finding database antipatterns in SQL query logs has not been studied before systematically. In this paper, we have proposed a solution for the detection of patterns and for solving antipatterns in such a log. To this end, we have formalized the notion of pattern in the current context. Next, we have provided rules for detecting and – if possible – solving antipatterns. Properties of patterns and antipatterns allow the discovery of certain kinds of user behavior, as a case study on the SkyServer query log has demonstrated. All in all, our approach is capable to detect and classify patterns in a query log. The

results show a significant number of instances of antipatterns within the log. Moreover, it is feasible to remove the most frequently occurring antipatterns. The remaining patterns refer to real user information needs. All this highlights the importance of the approach as a general preprocessing step for any subsequent SQL log analysis.

Regarding the influence of cleaning the learning set on the quality of query recommendation, our outline for future work is as follows:

1. We hypothesize that sliding-window search (SWS) bogs down the quality of queries recommended. Namely, since SWS comes from robots, the queries might differ in nature from ones formulated by mortal users. When including SWS in the learning set, we will generate recommendations which rely on SWS as well.
2. Clearly, queries suggested by a recommender system must not contain antipatterns. We would like to study the rate of recommended queries containing antipatterns if the recommender is trained on the original log. We then would like to do the same with the cleaned log. If the rate now is much smaller, then our approach obviously is more useful compared to the outcome that it is not.

REFERENCES

[1] H. V. Nguyen et al., "Identifying User Interests within the Data Space – a Case Study with SkyServer," in *EDBT*, Brussels, 2015.

[2] W. Brown, AntiPatterns: refactoring software, architectures, and projects in crisis, Wiley, 1998.

[3] Gamma, Erich, Design patterns: elements of reusable object-oriented software, Pearson Education India, 1995.

[4] B. Dudney et al., J2EE antipatterns, Wiley, 2003.

[5] M. Fowler et al., Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[6] J. Akbarnejad et al., "SQL QueRIE recommendations," *VLDB Endowment*, vol. 3, no. 2, 2010.

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

[8] H. Cao, "Context-Aware Query Suggestion by Mining Click-Through," in *KDD*, 2008.

[9] V. Singh et al., "SkyServer Traffic Report—The First Five Years," Microsoft Research, 2006.

[10] M. Jordan Raddick et al., "Ten Years of SkyServer I: Tracking Web and SQL e-Science Usage," *Computing in Science and Engineering*, vol. 16(4), 2014.

[11] M. Jordan Raddick et al., "Ten Years of SkyServer II: How Astronomers and the Public Have Embraced e-Science," *Computing in Science and Engineering*, vol. 16(4), 2014.

[12] X.Wang, A. Meliou, E. Wu, "QFix: Diagnosing errors through query histories," *eprint arXiv:1601.07539*, 2016.

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

[14] E. Eessaar, "On Query-Based Search of Possible Design Flaws of SQL Databases," *Springer*, vol. 313, 2014.

[15] T-H. Chen et al., "Detecting Performance Anti-patterns for Applications Developed using Object-Relational Mapping," in *ICSE*, 2014.

[16] T-H Chen et al., "Detecting problems in the database access code of large scale systems: an industrial experience report," in *ICSE*, 2016.

[17] A. Wert et al., "Automatic Detection of Performance Anti-patterns in Inter-component Communications," in *QoSA*, 2014.

[18] C. Trubiani, A. Koziolek, "Detection and solution of software performance antipatterns in Palladio architectural models," in *ICPE*, 2011.

[19] C. Smith, L. Williams, "Software performance antipatterns," in *WOSP*, 2000.

[20] F. Buschmann et al., Stal: Pattern-Oriented Software Architecture - A System of Patterns, John Wiley & Sons, 1996.

[21] I.D. Baxter et al., "Clone Detection Using Abstract Syntax Trees," in *ICSM*, 1998.

[22] S. Schulze, S. Apel, C. Kästner, "Code clones in feature-oriented software product lines," in *GPCE*, 2010.

[23] J. Jones, "Abstract Syntax Tree Implementation Idioms," 2010.

[24] B. Karwin, SQL Antipatterns. Avoiding the Pitfalls of Database Programming, Pragmatic Bookshelf, 2010.

[25] D. Burleson, V. Tropashko, SQL Design Patterns: Expert Guide to SQL Programming, Rampant TechPress, 2007.

Natalia Arzamasova has received her diploma degree from Chuvash State University (Russia), worked as a software engineer in Vocord Software Company, creating automatic enterprise databases, services and user interface. Currently she is working in Karlsruhe Institute of Technology (KIT), Germany, on her Ph.D. Her research interests include query log analysis.

Martin Schäler received his Master degree from Otto-von-Guericke University Magdeburg, Germany, in 2010. Afterwards he was employed as a research assistant and scientific coordinator at the same university, receiving his Ph.D. degree in 2014. Since August 2015 he is a post-doctoral researcher at the databases and information systems group of Karlsruhe Institute of Technology (KIT), Germany. His research interests include multi-dimensional access methods, hardware-sensitive database tuning and provenance.

Klemens Böhm is full professor (chair of databases and information systems) at Karlsruhe Institute of Technology (KIT), Germany, since 2004. Prior to that, he has been affiliated with University of Magdeburg, Germany, ETH Zurich, Switzerland, and GMD – Forschungszentrum Informationstechnik GmbH, Darmstadt, Germany. Current research topics at his chair are knowledge discovery and data mining in big data, data privacy and workflow management.