SQL query log analysis for identifying user interests and query recommendations

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Natalia Arzamasova

Tag der mündlichen Prüfung: 03.02.2020

Erster Gutachter: Herr Prof. Dr. Klemens Böhm

Zweiter Gutachter: Herr Prof. Dr. Michael Grossniklaus

Abstract

In the sciences and elsewhere, the use of relational databases has become ubiquitous. To get maximum profit from a database, one should have in-depth knowledge in both SQL and a domain (data structure and meaning that a database contains). To assist inexperienced users in formulating their needs, SQL query recommendation system (SQL QRS) has been proposed. It utilizes the experience of previous users captured by SQL query log as well as the user query history to suggest. When constructing such a system, one should solve related problems: (1) clean the query log and (2) define appropriate query similarity functions. These two tasks are not only necessary for building SQL QRS, but they apply to other problems. In what follows, we describe three scenarios of SQL query log analysis: (1) cleaning an SQL query log, (2) SQL query log clustering when testing SQL query similarity functions and (3) recommending SQL queries. We also explain how these three branches are related to each other.

Scenario 1. Cleaning SQL query log as a general pre-processing step

The raw query log is often not suitable for query log analysis tasks such as clustering, giving recommendations. That is because it contains antipatterns and robotic data downloads, also known as Sliding Window Search (SWS). An antipattern in software engineering is a special case of a pattern. While a pattern is a standard solution, an antipattern is a pattern with a negative effect.

When it comes to SQL query recommendation, leaving such artifacts in the log during analysis results in a wrong suggestion. Firstly, the behaviour of "mortal" users who need a recommendation is different from robots, which perform SWS. Secondly, one does not want to recommend antipatterns, so they need to be excluded from the query pool. Thirdly, the bigger a log is, the slower a recommendation engine operates. Thus, excluding SWS and antipatterns from the input data makes the recommendation better and faster.

The effect of SWS and antipatterns on query log clustering depends on the chosen similarity function. The result can either (1) do not change or (2) add clusters which cover a big part of data. In any case, having antipatterns and SWS in an input log increases only the time one need to cluster and do not increase the quality of results.

Scenario 2. Identifying User Interests via Clustering

To identify the hot spots of user interests, one clusters SQL queries. In a scientific domain, it exposes research trends. In business, it points to popular data slices which

one might want to refactor for better accessibility. A good clustering result must be precise (match ground truth) and interpretable.

Query similarity relies on SQL query representation. There are three strategies to represent an SQL query. FB (feature-based) query representation sees a query as structure, not considering the data, a query accesses. WB (witness-based) approach treat a query as a set of tuples in the result set. AAB (access area-based) representation considers a query as an expression in relational algebra. While WB and FB query similarity functions are straightforward (Jaccard or cosine similarities), AAB query similarity requires additional definition. We proposed two variants of AAB similarity measure – overlap (AABovl) and closeness (AABcl). In AABovl, the similarity of two queries is the overlap of their access areas. AABcl relies on the distance between two access areas in the data space – two queries may be similar even if their access areas do not overlap.

The extensive experiments consist of two parts. The first one is clustering a rather small dataset with ground truth. This experiment serves to study the precision of various similarity functions by comparing clustering results to supervised insights. The second experiment aims to investigate on the interpretability of clustering results with different similarity functions. It clusters a big real-world query log. The domain expert then evaluates the results. Both experiments show that AAB similarity functions produce better results in both precision and interpretability.

Scenario 3. SQL Query Recommendation

A sound SQL query recommendation system (1) provides a query which can be run directly, (2) supports comparison operators and various logical operators, (3) is scalable and has low response times, (4) provides recommendations of high quality. The existing approaches fail to fulfill all the requirements. We proposed DASQR, scalable and data-aware query recommendation to meet all four needs. In a nutshell, DASQR is a hybrid (collaborative filtering + content-based) approach. Its variations utilize all similarity functions, which we define or find in the related work.

Measuring the quality of SQL query recommendation system (QRS) is particularly challenging since there is no standard way approaching it. Previous studies have evaluated the results using quality metrics which only rely on the query representations used in these studies. It is somewhat subjective since a similarity function and a quality metric are dependent. We propose AAB quality metrics and then evaluate each approach based on all the metrics.

The experiments test DASQR approaches and competitors. Both performance and runtime experiments indicate that DASQR approaches outperform the existing ones.

Deutsche Zusammenfassung

In der Wissenschaft und anderswo ist die Verwendung von Datenbanken allgegenwärtig geworden. Um den maximalen Nutzen aus einer Datenbank zu ziehen sollte man eingehendes Wissen haben, sowohl von SQL als auch von der Domäne (eine Datenstruktur, die eine Datenbank enthält). Um unerfahrene Benutzer beim Formulieren Ihrer Informationsbedürfnisse zu unterstützen wurden SQL-Anfrage-Empfehlungssysteme (SQL QRS) vorgeschlagen. Diese verwenden sowohl die Erfahrungen früherer Nutzer, die in SQL-Anfrage-Logs erfasst werden, als auch die Anfragehistorie des Benutzers um Vorschlage zu erzeugen.

Beim Konstruieren eines solchen Systems sollten die folgenden Probleme lösen werden: (1) bereinigen des Anfrage-Logs und (2) geeignete Funktionen zum Quantifizieren der Anfrageähnlichkeit definieren. Diese zwei Aufgaben sind nicht bloß notwendig zum Bau eines SQL QRS, sondern gelten auch für andere Probleme. Im Folgenden beschreibe ich drei Szenarios der Analyse von SQL-Anfrage-Logs: (1) bereinigen des SQL-Anfrage-Logs, (2) clustern von SQL-Anfrage-Logs beim Testen der Ähnlichkeitsfunktionen von SQL-Anfragen und (3) vorschlagen von SQL-Anfragen. Im Folgenden werde ich detaillierter Beschreiben wie diese drei Teilgebiete miteinander verbunden sind.

Szenario 1. Bereinigen des SQL-Anfrage-Logs als allgemeiner Vorverarbeitungsschritt

Das unbearbeitete Anfrage-Log eignet sich häufig nicht für Analyseaufgaben, wie Clustern oder Empfehlungen geben. Das liegt daran, dass es Anti-Pattern und Datendownloads von Robotern enthält, die auch als Sliding Window Search (SWS) bezeichnet werden. In der Softwareentwicklung ist ein Anti-Pattern ein Sonderfall eines Pattern: Während ein Pattern eine Standardlösung ist, ist ein Anti-Pattern ein Pattern mit einem negativen Effekt.

Wenn im Falle von SQL QRS solche Artefakte während der Analyse im Log bleiben, kann das zu einer falschen Empfehlung führen. Erstens unterscheidet sich das Verhalten „sterblicher“ Benutzer, die eine Empfehlung benötigen, von Robotern, die SWS ausführen. Zweitens möchte man keine Anti-Pattern empfehlen, daher müssen diese aus dem Anfragepool entfernt werden. Drittens arbeitet eine Empfehlungsengine umso langsamer, je größer das Log ist. Wenn SWS und Anti-Pattern aus den Logs entfernen werden, macht das die Empfehlung besser und schneller.

Die Auswirkung von SWS und Anti-Pattern auf das Clustering von Anfragelogs hängt von der gewählten Ähnlichkeitsfunktion ab. Entweder kann sich das Ergebnis
(1) nicht ändern oder (2) es werden Cluster hinzugefügt, die einen großen Teil der Daten

abdecken. Im ersten Fall verlängert sich die Zeit zum Clustern, wenn Anti-Pattern und SWS in den Eingabelogs enthalten sind. Im zweiten Fall erhält man Cluster, die nicht die Hotspots der Benutzerinteressen zeigen.

Szenario 2. Identifikation der Benutzerinteressen durch Clustering

Um Ähnlichkeitsfunktionen für Anfragen zu finden und zu testen, werden SQL-Anfragen geclustert. Das Clustern ist eine eigenständige Aufgabe, die Benutzerinteressen offenbart. Im wissenschaftlichen Bereich werden damit Forschungstrends aufgedeckt. Im Unternehmensbereich weist es auf beliebte Teildaten, die für bessere Zugänglichkeit überarbeitet werden könnten. Ein gutes Cluster-Ergebnis muss präzise (der Ground-Truth entsprechend) und interpretierbar sein.

Die Anfrageähnlichkeit hängt von der Darstellung der SQL-Anfrage ab. Es gibt drei Strategien SQL-Anfragen darzustellen. Der FB-Ansatz (Feature-basiert) sieht eine Anfrage als eine Struktur, ohne Berücksichtigung der Daten, auf die eine Anfrage zugreift. Der WB-Ansatz (Zeugen-basiert) repräsentiert eine Anfrage als eine Menge von Tupeln in der Ergebnismenge. Der AAB-Ansatz (Anfragebereich-basiert) betrachtet eine Anfrage als Ausdruck in der relationalen Algebra. Während die Ähnlichkeitsfunktionen von WB- und FB-Anfragen für die ersten beiden Ansätze (Jaccard- oder Cosinus-Ähnlichkeit) unkompliziert sind, erfordert die Ähnlichkeit von AAB-Anfragen eine zusätzliche Definition. Ich habe zwei Varianten des AAB-Ähnlichkeitsmaßes vorgeschlagen — Überlappung (AABovl) und Nähe (AABcl). Die Ähnlichkeit von zwei Anfragen in AABovl ist die Überlappung ihrer Zugriffsbereiche. AABcl basiert auf dem Abstand zwischen den beiden Zugriffsbereichen im Datenraum — zwei Anfragen können ähnlich sein, auch wenn sich Ihre Zugriffsbereiche nicht überlappen.

Die umfangreichen Experimente bestehen aus zwei Teilen. Das erste Experiment ist das Clustern eines eher kleinen Datensatzes anhand der Ground-Truth. Dieses Experiment dient dazu die Genauigkeit verschiedener Ähnlichkeitsfunktionen zu untersuchen indem die Ergebnisse des Clusterings mit überwachten Erkenntnissen verglichen werden. Das zweite Experiment zielt darauf ab, die Interpretierbarkeit der Cluster-Ergebnisse mit verschiedenen Ähnlichkeitsfunktionen zu untersuchen. Dieses clustert ein großes Echtwelt-Anfrage-Log. Ein Domänen-Experte bewertet anschließend die Ergebnisse. Beide Experimente zeigen, dass AAB-Ähnlichkeitsfunktionen bessere Ergebnisse liefern, sowohl bei der Genauigkeit als auch bei der Interpretierbarkeit.

Szenario 3. SQL Anfrageempfehlung

Ein zuverlässiges SQL-Anfrageempfehlungssystem (1) liefert eine Anfrage, die direkt ausgeführt werden kann, (2) unterstützt Vergleichsoperatoren und verschiedene logische Operatoren, (3) ist skalierbar und weist niedrige Antwortzeiten auf und (4) liefert Empfehlungen von hoher Qualität. Existierende Ansätze erfüllen nicht alle diese Anforderungen. Ich habe DASQR vorgeschlagen, eine skalierbare und datensensitive Anfrageempfehlung, die alle vier Anforderungen erfüllt. Kurz gesagt ist DASQR ein hybrider Ansatz bestehend aus kollaborativem Filtern und einem inhaltsbasierten

Ansatz. Seine Varianten nutzen alle Ähnlichkeitsfunktionen, die ich definiert oder in verwandten Arbeiten gefunden habe.

Das Messen der Qualität von SQL QRS ist eine besondere Herausforderung, da es keine Standardmethode gibt. Bisherige Studien haben die Ergebnisse anhand von Qualitätsmetriken ausgewertet, die auf Anfragerepresentationen basieren, die nur in diesen Studien verwendet wurden. Das ist etwas subjektiv, da Ähnlichkeitsfunktion und Qualitätsmetrik unabhängig voneinander sind. Ich schlage AAB-Qualitätsmetriken vor und bewerte jeden Ansatz basierend auf allen Qualitätsmetriken.

Die Experimente testen meinen DASQR-Ansatz gegen Wettbewerber. Sowohl Performance- als auch Laufzeitexperimente zeigen, dass mein DASQR-Ansatz existierende Ansätze übertreffen.

Acknowledgements

Here I would like to thank all the people who supported me during the writing of this thesis.

First of all, I'm very thankful to my supervisor Klemens Böhm. Next, thanks to my colleagues for advises in my research: Vadim Arzamasov, Georg Steinbuss, Holger Trittenbach, Jens Willkomm, Edouard Fouché, Michael Vollmer and Fabian Laforet.

Many thanks again to Vadim Arzamasov for making me finish it. Last but not least, I thank Varvara Arzamasova for her patience and understanding.

Table of Contents

1 Introduction

SQL query recommendation suggests an SQL query to a user, based on his submitted requests and the ones of other users stored in a query log. When constructing such system, there are related problems appear: (1) clean a query log and (2) define appropriate query similarity functions. These two tasks do not only serve for developing SQL QRS, but apply to independent problems. In what follows, we describe the three scenarios of SQL query log analysis, namely (1) cleaning an SQL query log (2) SQL query log clustering when testing SQL query similarity functions and (3) recommending SQL queries. Moreover, we explain the relationship between these scenarios.

1.1 Cleaning SQL query log as general pre-processing step

To cluster or give SQL query recommendation, one should first clean a log from antipatterns and robotic data downloads a.k.a. Sliding Window Search (SWS). An antipattern in software engineering is a special case of a pattern. While a pattern is a common solution, an antipattern is a pattern with a negative effect.

Example 1.1 Table 1.1 lists a sequence of SQL queries of a user. These statements reflect the 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 that constant. That is an occurrence of the Circuitous Treasure Hunt (CTH) antipattern [Dud+03]. Next, the second and the third query select different columns of the same table. That is the Stifle antipattern [Fow97].

A common method to detect antipatterns requires access to the software that generates the requests [Dud+03]. Regarding SQL antipatterns, this means that one would need

Table 1.1: 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	03/12/1985, 01259863448
4	**SELECT** count(orders) **FROM** Orders O **WHERE** O.empId = 12	36

to have access to all systems working with the database. That 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 1.1 has mentioned, 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 re-run the first query and inspect the result. However, this is not viable for the following reasons:

(1) *Performance aspect*: Re-running a significant part of an SQL log implies a huge load on the database.
(2) *Side effects aspect*: 'Re-run' queries will be saved 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.

Cleaning antipatterns on an SQL query log is also a general preprocessing step in the data-analysis process. Clean query log serves as an input for follow-up query analysis tasks. Cleaning influences them in the following way.

1.1.1 The influence of SQL query log cleaning to finding user interests via clustering

Let us demonstrate the effect of antipatterns cleaning when it comes to finding hot spots of user interests on the examples.

Example 1.2 Queries 2 and 3 of Example 1.1 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 1.3 Consider again Table 1.1. Queries 2 to 4 can only be understood together with Query 1. That is because the Attribute id does not have any meaning from the domain perspective. Thus, if one rewrites queries without antipatterns, the specific user interest would be more obvious:

```
SELECT E.empId, E.name, E.surname, 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 effect of SWS on query log clustering depends on the chosen similarity function. The result can either (1) do not change or (2) add clusters which cover a big part of data. In the first case having antipatterns and SWS in an input increases the time one need to cluster. In the second case, one gets clusters, which do not show hot spots of user interests.

1.1.2 The influence of SQL query log cleaning to SQL query recommendations

When it comes to SQL QRS, leaving SWS and antipatterns in the log during analysis may result in a wrong recommendation. Firstly, the behaviour of "mortal" users, who need advice, is different from robots, which perform SWS. Secondly, one does not want to recommend antipatterns, so they need to be excluded from the query pool. Thirdly, the bigger a log is, the slower a recommendation engine operates. Thus, excluding SWS and antipatterns from the log makes the recommendation better and faster.

1.2 Identifying User Interests via clustering

An SQL query is a request for data or information from a database table or combination of tables. This definition is rather abstract and does not answer the question "how to compare two queries?". The second topic of our interest is finding an optimal way to represent and compare SQL queries.

To test query similarities, we cluster SQL queries. By doing so, we aim to reveal user interests. In a scientific domain, user interests expose research trends. In business, it points to popular data slices, which one might want to refactor for better accessibility. We are guided by two criteria while evaluating the clustering results: *precision* and *interpretability* [HBV01]. A precise cluster matches the reality (or ground truth). The interpretable cluster represents one clear user interest. [HBV01].

We see three main challenges on the way of clustering SQL query log:

(1) To cluster SQL queries, one needs a notion of query similarity. Query similarity relies on SQL query representation. There are three strategies to represent an SQL query:
    (a) *FB (feature-based) approach* [Kho+10] sees a query as structure, not considering the data a query access.
    (b) *WB (witness-based) approach* [Eir+14] represents a query as a set of tuples in the result set.
    (c) *AAB (access area-based) approach* [Ngu+15] considers a query as an expression in the relational algebra.

Table 1.2: Queries in a log

#	Statements	Result
1	**SELECT** * **FROM** Employees E **WHERE** E.department = 'sales'	12 employees from sales department
2	**SELECT** * **FROM** Employees E **WHERE** E.department = 'store'	8 employees from store department
3	**SELECT** * **FROM** Employees E **WHERE** E.startdate >= '01/12/2015'	10 employees who started working in a company after the date

Example 1.4 Think of a query log consisting of the queries listed in Table 1.2. All three queries access table 'Employees'. One might find the first and the second query similar. That is because they have the same structure, asking for employees in a particular department. Indeed, according to the *FB* approach, these two queries are identical. However, one can also disagree with this conclusion. In line with the *WB* representation, these two queries do not have any common tuples in their results sets. When it comes to *AAB*, the first and the second query refer to different parts of the data space and hence are not similar. Regarding the similarity between the first and the third query, one cannot say much. Even though there could be employees from the sales or the store department who started to work after 01/12/2015 (similarity in *WB*), this does not lead to meaningful insights. A user might have had different intentions when formulating these queries.

So far, to our knowledge, there is no comparative study on the usefulness of different query representations for clustering.

(2) Once one has a query representation, he can build a similarity function on it. A query similarity function quantifies for any two queries to what extent they are alike. The *FB* and *WB* representations lend themselves to straightforward measures: Jaccard or cosine similarity. The similarity function for *AAB* in turn proposed in [Ngu+15] is complex compared to *FB* or *WB*. It also has some redundancies, and several definitions behind it are ad hoc, as we will explain (see Section 3.3.4). Generally speaking, we also wonder whether there are more answers to the question when two queries are similar.

(3) Another challenge when testing similarity function via clustering is that we are not aware of any suitable publicly available data set, including a ground truth. Ground truth is needed to quantify the *precision* of clustering. A 'suitable' query log must include (a) a labeled SQL query log and (b) the database these queries have been submitted to. It also (c) must be publicly available. So one cannot objectively compare similarity functions and the corresponding query representations.

Table 1.3: Queries q_1 and q_2 submitted in a row

#	Statements
1	`SELECT obj FROM Galaxies G` `WHERE G.ra BETWEEN 29.122 AND 29.01 AND G.dec BETWEEN 0.996 AND 1.103`
2	`SELECT obj FROM Galaxies G` `WHERE G.ra BETWEEN 29.065 AND 29.066 AND G.dec BETWEEN 1.016 AND 1.017`

1.3 SQL Query Recommendation

Open-access databases let users retrieve data online. SQL serves as a common interface to this end. However, it often is not helpful for users with little database expertise. SQL query recommendation systems (SQL QRS) have been proposed to deal with this mismatch [Kho+10], [Ali+15], [RRS11a].

A clean (without antipatterns and SWS) query log is as an input of building such a system. The query similarity functions that we discover or define while identifying user interest via clustering (Section 1.2) serve as one of building bricks on the way.

We formulate the following requirements to SQL query recommendation system:

(1) provides a query which can be issued directly,
(2) supports comparison operators and various logical operators,
(3) is scalable and has low response times,
(4) provides recommendations of high quality.

We now explain and motivate these requirements.

(1) *Recommending full and data-aware queries.* 'Full query' means suggesting an entire SQL request, not only a segment (like tables in the FROM clause). Data awareness implies having real values (not placeholders) in the filtering condition of the query. A full and data-aware request can be submitted to a database as is.

    Example 1.5 Table 1.3 contains two queries submitted in a row, to the SkyServer database[*]. q_1 searches for galaxies in a cluster[†]. q_2 zooms into a particular area within the cluster. While it is not difficult to put together the structure of the query, the complication rather is specifying the area to zoom into. Here, suggesting only a query structure or a part of it is less helpful.

(2) *Support of comparisons and various logical operators.* Comparison operators of SQL are $\{=, <, >, \leq, \geq, \neq\}$. Popular logical operators are ALL, AND, ANY, BETWEEN, EXISTS, IN, NOT, OR, SOME. As motivated in Example 1.5, range queries are needed. Our literature review will reveal that existing OLAP-oriented approaches do not cover them.

[*] http://skyserver.sdss.org
[†] A galaxy cluster is a structure that consists of anywhere from hundreds to thousands of galaxies that are bound together by gravity. (https://www.spacetelescope.org/news/heic1201/)

Table 1.4: The recommended q_1 and the unseen q_2 queries

#	Statements	Result
1	`SELECT name, population FROM Cities C WHERE C.population BETWEEN 2000 AND 4000`	{Rome, 2863}; {Madrid, 3223}
2	`SELECT name, population FROM Cities C WHERE C.population BETWEEN 1000 AND 2500`	{Vienna, 1899}; {Minsk, 1982}

(3) *Scalability and speed.* 'Low response time' means that a recommendation is generated within a few seconds and that this time grows slower than linearly with the log size.

(4) *High quality.* In a nutshell, recommendations need to be helpful to the user.

The last requirement is particularly challenging since there is no standard way to measure the quality of query recommendations. In principle, one would need to compare a recommended query to an *unseen* one, the query that represents the information need of a user (if it was known). The result of such a comparison would depend on the query representation.

Example 1.6 Table 1.4 contains two queries, q_1 and q_2, where q_1 is a recommended query and q_2 an unseen one. If we see a query as a structure, the two queries are identical. If we compare queries based on their result sets, q_1 and q_2 have nothing in common. So the outcome of comparison depends on the query representation.

We have motivated data-aware recommendations; they are essential in many situations. Such quality measures like precision and recall must capture the differences between the recommended and the unseen queries in the data space. However, the problem is that previous studies have evaluated their results using quality metrics which only rely on the query representations they have used themselves. For instance, [CEP09] relies on the witness-based (WB) representation, where the result tuples stand for the query. The approach then uses the notion of witness in its quality metrics as well. Evaluations, where the query representation and the quality metrics are decoupled, is missing, and is expected to provide interesting insights.

To our knowledge, no existing approach satisfies all of the above requirements, see Table 3.4. We aim to design such a recommendation system.

1.4 Main Contributions

Our first contribution is a general solution of detecting and solving antipatterns in an SQL query log. In particular,

(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. We also 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 4.2.4).
(3) Our empirical study relies on the log of Sky-Server system, covering seven 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 patterns among the 15 most frequent ones are antipatterns. After removing them, all patterns among the 40 most frequent ones do represent important information needs.
(5) We present evidence that the results of previous studies of the SkyServer log (e.g., [Ngu+15], [CEP09]) would have been different, had the log been cleared of antipatterns.

Our second contribution is studying the usefulness of SQL query similarity measures to find user interests. Throughout this process,

(1) We provide an extensive discussion of existing measures for query similarity and their advantages and disadvantages.
(2) Based on this discussion, we propose a new query similarities.
(3) We perform systematic experiments with the design alternatives. In particular, we study the impact of various similarity functions and query representations on clustering quality.
(4) To quantify the *precision* of clustering, data with ground truth is needed. Having such data in our current domain is an issue that existing approaches have difficulties with. We, in turn, come up with conditions where one knows in advance which cluster a query belongs to. Then we collect these queries together with this ground truth. We make this data publicly available.
(5) To measure *interpretability* we conduct a study which involves a domain expert. He interprets various clustering results and assesses how well they align with user interests.
(6) We find that our proposed similarity measures are better than the existing ones regarding both *precision* and *interpretability* and provide explanations for this. We have learned that the new measures are indeed helpful to arrive at 'query clusters' that are meaningful, i.e., represent user interests.

Our third contribution is DASQR, a data-aware and scalable SQL query recommendation system. More specifically,

(1) We put together systematically which design decisions are necessary when constructing an SQL QRS and then look at the respective solutions of existing systems. More specifically,

  (a) We propose both content-based (CB) and collaborative filtering (CF) versions of SQL QRS, as well as hybrid combinations. However, there are significant differences between the SQL query domain and, say, e-commerce, and instantiating these principles for SQL is a contribution of ours.

  (b) We have designed CB query recommendation to study how beneficial it is to recommend queries similar to already submitted ones. Previous studies do not feature CB SQL query recommendations.

  (c) Regarding collaborative filtering (CF), we come up with several proposals: A first one (TkS) operates directly in the data space, a second one (TnS) first finds a suitable structure of the recommended query and then assigns filtering conditions. We also propose a hybrid approach and explain that, though simple, it is more specifically tailored to the SQL domain than earlier hybrid proposals.

  (d) Some of our design decisions are borrowed from existing approaches which are plausible and convincing. For instance, we make use of existing query similarity functions (AABovl, AABcl) [Arz+19] and of other existing features (FB [Kho+10], WB [CEP09]). However, regarding user sessions, while we have started with existing methods (SWA [SW+81], DTW [Sak+90]) to quantify their similarity, a proposal of our own has turned out to be better.

(2) We address the problem of existing solutions that quality metrics only rely on the current representation. We introduce measures for all representations considered and then compare the recommendations according to all measures. The rationale is to learn more about the results if a metric does not correspond to a query representation. In our experiments, we study whether the results are stable irrespective of the SQL query representation the quality metric is built upon.

(3) An extensive experimental study shows that our approaches outperform the competitors irrespective of the quality measure.

1.5 Outline

Chapter 2 presents basic definitions and concepts that we use in this thesis. Next, Chapter 3 presents an overview of existing techniques and algorithms related to SQL query log analysis. In particular, it studies SQL antipatterns, SQL query similarity functions and SQL query recommendations. Chapter 4 presents our cleaning solution for SQL query logs. Afterwards, we present and test our query similarity functions in Chapter 5. In Chapter 6 we propose DASQR, our data aware SQL query recommendation approach. The discussion of the results is presented in the end of each corresponding chapter as well as in Chapter 7.

2 Preliminaries

In this chapter, we present basic definitions and concepts that are used throughout this thesis.

2.1 Common Definitions

Definition 2.1 A relational database DB is a database consisting of N relations $R_1, \dots, R_N$.

Definition 2.2 A database schema S is a logical architecture of the database DB, i.e., a set of definitions of relations $\{R_1, \dots, R_N\}$ of DB and constraints put on them.

Definition 2.3 A database state T of the database DB is data in the database DB allowed by the database schema S at any particular time.

Definition 2.4 $(U;T)$ stands for the set of tuples of U at state T of DB.

Definition 2.5 A query q is a Select-Project-Join (SPJ) request together with an optional aggregate computation.

Definition 2.6 A domain $dom(a)$ of an attribute a is one of an attribute a.

Definition 2.7 The universal relation U is the Cartesian product of all relations in the database $U = R_1 \times \cdots \times R_N$.

2.2 SQL Query

Definition 2.8 The universal relation $U(q)$ of a query q is the Cartesian product of all relations in the query $q : U(q) = R_1 \times \cdots \times R_M$.

Definition 2.9 Filtering conditions $P(q)$ of a query q is a Boolean expression of all constrains put on U by a query q.

Definition 2.10 The interest *Ints(q)* of a query q is the sequence of attributes (in alphabetical order) occurring in the filtering conditions $P(q)$.

Example 2.11 Let us define interests of a query q:

```
SELECT * FROM Employees E WHERE E.department = 'sales'
      AND E.startdate >= '01/12/2015' AND E.startdate <= '01/12/2018'
```

Ints(q)= (Employees.department, Employees.startdate).

Let us refer to a particular *interest* in $Ints(q)$ as *Ints(q)[i]*, starting an index i from 1. For instance, *Ints(q)[1]=Employees.departmentandInts(q)[2]=Employees.startdate.*

Definition 2.12 The operators *operators(q)* of a query q is a sequence of interests with connected operators in the filtering conditions $P(q)$.

Example 2.13 A query q from Example 2.11 has the following operators:

$$(\{Employees.department; \{'='\}\}, \{Employees.startdate; \{'>=', '<='\}\})$$

Definition 2.14 *Operators of the interest* $operators(q)[ints]$ or $operators(q)[i]$ are operators, which refer to the particular *interest ints*

Here i is an index of an interest *ints* within $Ints(q)$.

Example 2.15 *Operators of the interest* of Query q and *interest Employees.startdate* from Example 2.11 are:

$$operators(q)[Employees.startdate] = operators(q)[2] = \{'>=', '<='\}.$$

Definition 2.16 *Filtering conditions of the interest* $P(q)[ints]$ or $P(q)[i]$ are ones, which refer to the particular *interest ints.*

Example 2.17 Let us define filtering conditions of interests for q from Example 2.11:

$$P(q)[Employees.department] = P(q)[1] = Employees.department =' sales';$$

$$P(q)[Employees.startdate] = P(q)[2] =$$
$$\{Employees.startdate >=' 01/12/2015',$$
$$Employees.startdate <=' 01/12/2018'\}.$$

2.3 SQL Query Log

Users may submit queries in a sequence with a short time between them, so-called *user sessions.*

Definition 2.18 A user session us_i is a sequence of queries $(q_1^i, \ldots, q_n^i)$ of one user.

Here $l = |us_i|$ is the *length* of us_i. A user session often reflects a certain intention of the user ([CEP09], [Eir+14], [Ali+15]).

Definition 2.19 A query log US is a set of user sessions $\{us_1, \ldots, us_n\}$.

2.4 SQL Query Representations

Definition 2.20 A representation scheme QRS of a Query q is a function which returns certain feature values representing a query.

Definition 2.21 A query representation $QR(q)$ of a query q is a set of feature values which are the result of QRS applied to q.

Example 2.22 Think of a query log consisting of the queries listed in Table 1.2.

If a QRS which extracts the tables listed in the FROM clause, but nothing else,

$$QR(q_1) = QR(q_2) = QR(q_3) = \mathit{Employees}.$$

In contrast, if the QRS also extracts the attributes listed in the WHERE clause,

$$QR(q_1) = QR(q_2) = \mathit{Employees},\ \mathit{Employees.department},$$
$$QR(q_3) = \mathit{Employees}, \mathit{Employees.startdate}.$$

2.5 SQL query recommendations

Definition 2.23 us_0 is the *current* user session, i.e., the one for which the recommendation is made.

Definition 2.24 An unseen query q_{n+1}^0 is the one intended, but not formulated by the user.

Unseen queries are contained in the respective session, i.e., $|us_0| = n+1$.

nr is a number of recommendations an SQL recommendation system should provide.

The problem of SQL query recommendation is to suggest a ranked list of queries $(q'_1, \dots q'_{nr})$ that predict q^0_{n+1}. The subscript i of q'_i is the rank of the respective suggestion. Queries with the small ranks are recommended first. $(q'_1, \dots q'_{nr})$ can be empty or only partially filled if an algorithm is unable to suggest queries for specific input data $\{US, us_0\}$.

3 Related Work

In this chapter, we give an overview of the SQL Query Log Analysis approaches. We split it into four sections:

(1) general SQL query log analysis,
(2) review of database antipatterns,
(3) SQL query representations and corresponding similarity functions,
(4) SQL query recommendations

The first section overviews trends in SQL query log analysis. The second studies database antipatterns. Here we present the detection rules and refactoring solutions from the literature. Our third topic of interest is SQL query representations and corresponding similarity functions. We list the limitations of existed approaches, justifying the need for ours. The forth section first overview general approaches in making recommendations and then reviews the methods of giving SQL query suggestions.

3.1 General SQL Query Log Analysis

Query-log analysis currently is a field of intensive study. An elementary distinction is between weblogs and SQL logs. Studies on weblog processing such as [Sil+09] and [Cao+08] tend to focus on an understanding of the user behaviour through their information-seeking activities. [Sil+09] studies web search engine optimization by mining past queries. [Cao+08] proposes a context-aware query recommendation approach by mining click-through and session data.

A promising branch of SQL log analysis is diagnosing and repairing data errors caused by erroneous updates. [WMW17] 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, finding user interests and diagnosing errors. Our study relates to all these objectives.

Another research area, clustering SQL queries to identify hot spots of users' interests, is studied in [Ngu+15]. It proposes a query-similarity metric based on the notion of so-called access areas. We discuss this notion in detail when reviewing query representations and similarity measures in Section 3.3. [CL07] uses query clustering to help users locate interesting results. It generates clusters over the data. Each cluster corresponds to one type of user preference. To perform clustering, the authors compare queries based on the results they return. As an outcome, they present a navigational tree over clusters generated in the first step to the user. He can then select the subset of clusters matching his needs.

The third research thread applies association-rule mining to a query log. [CEP09] describes an approach that generates SQL query recommendations online. It compares user sessions and recommends a query to a user based on SQL requests from similar sessions. The authors present an idea that similar user behaviour manifests itself in similar data these users access. [Ali+14] presents a different approach to the similarity of SQL user sessions. The paper focuses on OLAP sessions and introduces an order-sensitive model to compare them. That means that the order of queries within a session influences their similarity. The proposed method considers the filtering conditions of SQL requests in a limited way: Only equality predicates are allowed. Other related work [Kho+10] aims at auto-completion of a query, suggesting tables, views, UDFs, columns and predicates. It adjusts its recommendation to the context: The more of the SQL request the user has typed in, the more accurate is the suggestion provided. [YPS09] recommends join queries based on log analysis. It first extracts chains of joins with corresponding predicates from the training set. The algorithm then creates queries from a test set with only tables present in these queries as an input.

Studies of SQL logs mainly consider publicly available scientific databases. The most popular one is the log from Sloan Digital Sky Survey, a.k.a. Sky Server[*]. [Sin+07], [Rad+14a] and [Rad+14b] 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 [Sin+07] and ten years [Rad+14a], [Rad+14b] since SDSS SkyServer has gone online.

[*]http://skyserver.sdss.org/dr15/en/home.aspx is a homepage for Sky Server project, the query log can be downloaded from http://skyserver.sdss.org/log/en/traffic/sql.asp

3.2 Review of Database Antipatterns

In the following, we briefly review research on database antipatterns. [BG06] 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. That work is orthogonal to ours. Database antipatterns, which we aim to detect, are semantically correct queries. [Ees15] detects database design antipatterns by querying metadata tables. Such antipatterns reflect errors in the database schema, which is not the topic of our study. [Che+14] proposes a framework to detect object-relational mappings (ORM) performance antipatterns. It performs static code analysis and a rule-based approach. It is again, not suitable for our task since we operate only with a query log. [Che+16] studies detection of DML bug patterns. DML queries, however, are not in the focus of this research: we aim to clean a query log of SELECT statements to facilitate further analyses on it (i.e., what do database users find interesting in the database).

In what follows we focus on two antipatterns, which influence analysis of an SQL query log. These are the Stifle [Wer+14] and the Circuitous Treasure Hunt [SW00]. They are also known to be the main reason for antipattern-related performance degradations. Our explanation includes suggestions for their detection and refactoring. We use a term *'refactoring'*, not *'solving'* in this context, because we reserve a term *'solving'* when we talk about query log modification.

3.2.1 Stifle Antipattern

3.2.1.1 Definition

The Stifle antipattern consists of several queries containing similar SQL statements [Fow97]. The term *similar* does not have a formal definition. However, examples are queries being identical except for constants in the WHERE clause. Processing such queries may be a bottleneck and harm 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 3.1 The following code generating SQL statements illustrates the Stifle antipattern. Thus, every *Id* in the *itemList* causes a request to Table T.

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

In a log, the Stifle manifests itself as a sequence of similar statements.

3.2.1.2 Detection

[Wer+14] bases its' detection approach for the Stifle on measurements of a running instance of the software. It assumes that a high load is a high-probability indicator for the presence of antipatterns. A specific sign 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. [TK11] proposes heuristics for measurement-based detection of several antipatterns on source-code level. There, the Stifle antipattern manifests itself by many similar database requests. A Stifle is detected if two or more database requests from one user for service exist. Besides, the queries need to have the same structure, except for the values passed to the method. Thus, this approach requires access to the source code of the service. Also, it compares strings used in the source code, which then form the query.

3.2.1.3 Refactoring

We now review methods to rewrite instances of the Stifle antipattern. In software development, this is called refactoring. [WMW17] proposes the *Pack* refactoring. Their idea is to collect individual SQL statements and send them to the database in one batch.

Example 3.2 The Pack refactoring for Example 3.1 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 query log to facilitate meaningful analyses.

3.2.2 Circuitous Treasure Hunt (CTH)

3.2.2.1 Definition

The Circuitous Treasure Hunt (CTH) antipattern [SW00] has one similarity with the Stifle, as they both consist of several database requests. However, individual CTH queries depend on each other. That means that a subsequent query requires the result of previous ones as input.

3.2.2.2 Detection

Similarly to the Stifle, a huge database overhead is an indication for the CTH antipattern [Wer+14]. However, to identify any CTH, one requires either knowledge on the queries or the ability to trace the information flow. Detection of CTH in [TK11] 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 1.1 has shown, without having the results of the first query, one does not see dependencies of a sequence of SQL statements with certainty.

3.2.2.3 Refactoring

The solving solution for CTH in [SW00] depends on the stage of 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 [SBM96]. 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.

3.3 SQL Query Representations and Similarity Functions

In this section, we address how to define the similarity of queries. Since an SQL query may have a complex structure, this is not trivial. First one has to decide what to compare. Put differently, which query-representation scheme (QRS) to use? We provide an overview of the approaches we have encountered in the scientific literature. Table 3.2 summarizes the query representations and the corresponding similarity/distance functions reviewed in this section. The last two rows refer to the new AAB similarity functions we are about to propose.

While studying the literature, we have in mind that our final goal is to create a data-aware query recommendation. Query similarity is a crucial auxiliary step on the way.

3.3.1 Query as a String

Arguably, the most straightforward way to represent an SQL query statement is as a string. To calculate query similarity, one could use string-similarity measures [II08]. However, this hardly captures any specific features of SQL.

3.3.1.1 String query similarity

Example 3.3 Consider the following queries:

q_1: `SELECT * FROM Employees WHERE birthyear < 1980`

q_2: `SELECT * FROM Employers WHERE birthyear < 1980`

On the level of string similarities, these two queries have a very small difference – only one character. However, they access entirely different tables and therefore, probably should have a very small similarity.

Besides, SQL keywords (SELECT, FROM, WHERE, etc.) overstate the similarity since they occur in every SPJ request. A possible solution is to exclude such words from consideration. However, this does not do away with effects like the one from Example 3.3 – even without keywords, these two strings differ by only one symbol.

3.3.1.2 Discussion

Representing an SQL query as a string does not capture it in a meaningful way. Of course, string-based query similarity cannot spot the resemblance in data two query access.

3.3.2 Query as Features

To overcome some of the obstacles described in Section 3.3 and to give more attention to the structure of an SQL request, [Kho+10] proposes a query representation as follows. There, a query is a set of features, and features are:

(1) tables, views, UDFs in the FROM clause
(2) attributes in the SELECT clause
(3) predicates (without values) in the WHERE clause
(4) attributes in the GROUP BY clause.

Example 3.4 Going back to queries from Example 3.3, FB representation for q_1 is:

$$\{f_{*}^{SELECT}, f_{Employees}^{FROM}, f_{Employees.birthyear<<num>}^{WHERE}\}$$

A slightly modified FB query representation defined in [Eir+14]. A query q_i is a vector $\phi^i = (\phi_1^i, \ldots, \phi_k^i)$ whose cell ϕ_j^i contains a weight if the feature ϕ_j appears in q_i. k is the number of possible features. There are two ways (weighting schemes) of setting ϕ_j^i:

(1) *binary scheme*, where all features of a query have weight 1,
(2) *weighted scheme*, where a feature weight ϕ_j^i is equal to the number of times the feature ϕ_j occurs in the query q_i.

Hence, FB representation from [Kho+10] corresponds to binary scheme in [Eir+14]. Weighted scheme of [Eir+14] is a slight alteration of it. From now on we refer to*FB binary* and *FB weighted* when talk about FB query representations from [Eir+14].

3.3.2.1 FB query similarity

According to [Eir+14], depending on the weighting scheme used, the query similarity either is the Jaccard coefficient (binary) or cosine similarity (weighted).

$$S_{FB_{bin}}(q_1, q_2) = \frac{|\phi^1 \cap \phi^2|}{|\phi^1 \cup \phi^2|} \tag{3.1}$$

$$S_{FB_{weig}}(q_1, q_2) = \frac{\phi^1 \cdot \phi^2}{\|\phi^1\| \cdot \|\phi^2\|} \tag{3.2}$$

3.3.2.2 Discussion

The FB approach captures the structure of a query and does not have the disadvantages of string-based similarity. However, it is not clear why all feature types do have the same importance: For instance, is the FROM clause always as important as the SELECT? It seems promising to have weights of features, depending on their position in the query (FROM, WHERE or SELECT clause). Next, one feature generally depends on other ones: For example, columns in the SELECT clause and predicates in the WHERE clause depend on the table in the FROM clause. Additionally, there is the question of how to work with '*' in a SELECT statement. The main weakness of such a query representation, however, is that it does not consider the values in a filtering condition, i.e., data unaware.

3.3.3 Query as Result Tuples

Another query representation scheme [CEP09], [Mat+10] introduces the notion of witnesses and is called witness based (WB) approach. This type of query representation is data-oriented. Here, a query represents itself by the tuples it returns, a.k.a. *witnesses.* As with FB, a WB query q_i is a vector τ^i whose element τ^i_j represents the importance of the tuple τ_j as a witness for q_i. Analogously to [Eir+14], [CEP09] proposes two ways of setting the importance:

(1) *binary scheme*: $\tau^i_j = \begin{cases} 1 \text{ if } \tau_j \text{ is a witness} \\ 0 \text{ otherwise} \end{cases}$

(2) *result-based scheme*: $\tau^i_j = \begin{cases} 1/\left|ans(q_i)\right| \text{ if } \tau_j \text{ is a witness} \\ 0 \text{ otherwise} \end{cases}$

Here $ans(q_i)$ is the result of q_i.

3.3.3.1 WB query similarity

According to [CEP09], one uses the Jaccard coefficient with the *binary scheme.* With *result-based*, it is cosine similarity.

$$S_{WB_{bin}}(q_1, q_2) = \frac{\left|\tau^1 \cap \tau^2\right|}{\left|\tau^1 \cup \tau^2\right|} \tag{3.3}$$

$$S_{WB_{res}}(q_1, q_2) = \frac{\tau^1 \cdot \tau^2}{\|\tau^1\| \cdot \|\tau^2\|} \tag{3.4}$$

From now on, we call the similarities *WBBin* and *WBRes.*

3.3.3.2 Discussion

We see several issues with this approach, as follows:

(1) *Necessity to re-query the database.* To identify all witnesses, one must run the queries another time, leading to a huge load on the database. Next, even if this was not an issue, it spoils subsequent query-log analysis. That is because re-run queries are stored in the query log. Finally, due to possible updates of the database in the meantime, there is no guarantee that a query will have the same result as the first time.

(2) *Result set can be empty.* Two queries which do not return any data cannot be compared even though they may be identical.

(3) *Possible insignificance of witness sets.* Due to the declarative nature of SQL in particular, the same data can be obtained in many different ways. Consider again Example 2.22. It is possible for Queries q_1 and q_3 to have similar result sets. However, the intentions behind the two queries are different.

Summing up, the WB approach overcomes the disadvantages of FB. It is clear and easy to implement. However, it may not be exactly practical in particular when the number of queries is vast.

3.3.4 Query as an Access Area

[Ngu+15] proposes a way to overtake the disadvantages of WB approach. The authors represent a query using the notion of so-called access areas. From now on, AAB is short for 'access area based query representation'. The access area of a query captures the area of the data space that the user is interested in.

Definition 3.5 A tuple $t \in U$ is said to influence the result set $(U,T)P$ of a query q iff $(U \backslash t\}, T)P \neq (U,T)P$. If t is removed from U, the result set of q at state T will change.

Definition 3.6 The access area of a query q is the set of all tuples t contained in the universal relation U that influence the result set of q in some database state T allowed by the schema which satisfy Predicate P of conditions put on a query q:

$$\{t \in U : \exists \quad T \quad \text{allowed by } DB \quad \text{s.t.t influences } (U,T)P\} \tag{3.5}$$

In contrast to WB, the access area of a query does not rely on the current database state. In contrary, it describes these tuples as an expression in the relational algebra. Coming back to q_1 from Example 3.3, the access area of Query q_1 is $\sigma_{department='Sales'}(Employees)$. The notion leaves aside the SELECT clause of the query. It considers predicates and the FROM clause. [Ngu+15] describes how to compute access areas for simple, join, aggregate and nested queries.

3.3.4.1 AAB query similarity

[Ngu+15] proposes a query distance measure, based on the overlap of the access areas of the two queries:

$$D(q_1,q_2) = d_{tables}(q_1.FROM, q_2.FROM) + d_{conj}(q_1.WHERE, q_2.WHERE) \tag{3.6}$$

$$d_{tables}(q_1.FROM, q_2.FROM) = 1 - \frac{|q_1.FROM \cap q_2.FROM|}{|q_1.FROM \cup q_2.FROM|} \tag{3.7}$$

The predicate P is in conjunctive normal form (CNF), i.e., it is a conjunction of clauses, where each clause is a disjunction of literals. Hence, $d_{conj}(b_1;b_2)$ in Formula 3.6 means distance of conjunctions. It is calculated as:

$$d_{conj}(b_1;b_2) = \frac{\sum_{o_1\in b_1} \min_{o_2\in b_2} d_{disj}(o_1;o_2) + \sum_{o_2\in b_2} \min_{o_1\in b_1} d_{disj}(o_1;o_2)}{|b_1|+|b_2|} \tag{3.8}$$

where each $o_i \in b_i$ is a disjunction of Boolean expression(s), and $|b_i|$ is the number of disjunctions of b_i in Query $q_i.d_{disj}(o_1,o_2)$ is the distance of the disjunctions of o_1 and o_2. It is as follows:

$$d_{disj}(o_1;o_2) = \frac{\sum_{p_1\in o_1} \min_{p_2\in o_2} d_{disj}(p_1;p_2) + \sum_{p_2\in o_2} \min_{p_1\in o_1} d_{disj}(p_1;p_2)}{|o_1|+|o_2|} \tag{3.9}$$

where $p_1 \in o_1$ is an atomic predicate, and $|o_1|$ is the number of atomic predicates of o_1. The distances between predicates are:

(1) $d_{pred}(p_1,p_2) = 1 - \frac{overlap\,Width(a)}{domain\,Width(a)}$ if both predicates p_1 and p_2 refer to the same attribute a;

(2) $d_{pred}(p_1,p_2) = 1 - \frac{width(a_1)}{domain\,Width(a_1)} \times \frac{width(a_2)}{domain\,Width(a_2)}$ if both predicates p_1 and p_2 refer to different attributes a_1 and a_2.

$width(a)$ is the width of the interval in which Predicate P is true.

Example 3.7 A query log contains of the following queries:

q_1: `SELECT * FROM Cities C WHERE C.latitude >= 40 AND C.longitude < 90`

q_2: `SELECT * FROM Cities C WHERE C.population >= 3000 AND C.country = 'USA'`

q_3: `SELECT * FROM Cities C WHERE C.latitude >= 40 OR C.longitude < 90`

q_4: `SELECT * FROM Cities C WHERE C.population >= 3000 OR C.country = USA'`

Attributes take the following values:

$C.latitude \in [-90; 90]$, $C.longitude \in [-180; 180]$, $C.population \in [0; 20000]$,
$C.country \in \{Afghanistan, \ldots, Zimbabwe\}$,
$|\{Afghanistan, \ldots, Zimbabwe\}| = 250$.

Let us denote $\alpha = C.latitude \geq 40$; $\beta = C.longitude < 90$; $\gamma = C.population \geq 3000$; $\delta = C.country ='USA'$.

Let us calculate the following distances: $D(q_1, q_2)$, $D(q_3, q_4)$, $D(q_1, q_4)$.

$d_{pred}(\alpha, \alpha) = 1 - (90-40)/(90-(-90)) = 0.72;$
$d_{pred}(\beta, \beta) = 1 - (180-90)/(180-(-180)) = 0.75;$
$d_{pred}(\gamma, \gamma) = 1 - 3000/20000 = 0.85;$
$d_{pred}(\delta, \delta) = 1 - 1/250 = 0.996;$
$d_{pred}(\alpha, \beta) = 1 - (90-40)/(90-(-90)) \cdot (180-90)/(180-(-180)) = 0.9305;$
$d_{pred}(\alpha, \gamma) = 1 - (90-40)/(90-(-90)) \cdot 3000/20000 = 0.9583;$
$d_{pred}(\alpha, \delta) = 1 - (90-40)/(90-(-90)) \cdot 1/250 = 0.998;$
$d_{pred}(\beta, \gamma) = 1 - (180-90)/(180-(-180)) \cdot 3000/20000 = 0.962;$
$d_{pred}(\beta, \gamma) = 1 - (180-90)/(180-(-180)) \cdot 1/250 = 0.999;$

Thus, the distances of disjunctions are:

$d_{disj}(\alpha, \gamma) = d_{pred}(\alpha, \gamma) = 0.9583;$
$d_{disj}(\alpha, \delta) = d_{pred}(\alpha, \delta) = 0.998;$
$d_{disj}(\beta, \gamma) = d_{pred}(\beta, \gamma) = 0.962;$
$d_{disj}(\beta, \delta) = d_{pred}(\beta, \delta) = 0.999;$

$d_{disj}(\alpha, \alpha \vee \beta) = (min(d_{pred}(\alpha, \gamma), d_{pred}(\alpha, \beta))) + d_{pred}(\alpha, \alpha) +$
$d_{pred}(\alpha, \beta))/3 = (0.72 + 0.72 + 0.9305)/3 = 0.79;$

$d_{disj}(\alpha, \gamma \vee \delta) = (min(d_{pred}(\alpha, \gamma), d_{pred}(\alpha, \delta))) + d_{pred}(\alpha, \gamma) +$
$d_{pred}(\alpha, \delta))/3 = (0.9583 + 0.9583 + 0.998)/3 = 0.971;$

$d_{disj}(\beta, \alpha \vee \beta) = (min(d_{pred}(\beta, \beta), d_{pred}(\alpha, \beta))) + d_{pred}(\beta, \beta) +$
$d_{pred}(\alpha, \beta))/3 = (0.75 + 0.75 + 0.9305)/3 = 0.81;$

$d_{disj}(\beta, \gamma \vee \delta) = (min(d_{pred}(\beta, \gamma), d_{pred}(\beta, \delta))) + d_{pred}(\beta, \gamma) +$
$d_{pred}(\beta, \delta))/3 = (0.962 + 0.962 + 0.9995)/3 = 0.9745;$

$d_{disj}(\alpha \vee \beta, \gamma \vee \delta) = (min(d_{pred}(\alpha, \gamma), d_{pred}(\alpha, \delta)) + min(d_{pred}(\beta, \gamma),$
$d_{pred}(\beta, \delta)))/4 + (min(d_{pred}(\alpha, \gamma), d_{pred}(\beta, \gamma)) + min(d_{pred}(\beta, \gamma))$
$+ min(d_{pred}(\alpha, \delta), d_{pred}(\beta, \delta)))/4 =$
$(0.9583 + 0.962 + 0.9583 + 0.998)/4 = 0.96915;$

The next step is calculating $d_{conj}(b_1, b_2)$, $d_{conj}(b_3, b_4)$ and $d_{conj}(b_1, b_4)$:

$d_{conj}(b_1, b_2) = d_{conj}(\alpha \wedge \beta, \gamma \wedge \delta) =$
$((min(d_{pred}(\alpha, \gamma), d_{pred}(\alpha, \delta)) + min(d_{pred}(\beta, \gamma), d_{pred}(\beta, \delta))))/4 +$
$((min(d_{pred}(\alpha, \gamma), d_{pred}(\beta, \gamma)) + min(d_{pred}(\alpha, \delta), d_{pred}(\beta, \delta))))/4 =$
$(0.9583 + 0.962 + 0.9583 + 0.998)/4 = 0.969;$

$$d_{conj}(b_3, b_4) = d_{conj}(\alpha \vee \beta, \gamma \vee \delta) = d_{disj}(\alpha \vee \beta, \gamma \vee \delta) = 0.96915;$$

$$d_{conj}(b_1, b_4) = d_{conj}(\alpha \wedge \beta, \gamma \vee \delta) = d_{disj}(\alpha, \gamma \vee \delta) + d_{disj}(\beta, \gamma \vee \delta))/3 + (min(d_{disj}(\alpha, \gamma \vee \delta), d_{pred}(\beta, \gamma \vee \delta)))/3 = (0.971 + 0.9745 + 0.971)/3 = 0.972;$$

$$d_{tables}(q_1.FROM, q_2.FROM) = d_{tables}(q_1.FROM, q_4.FROM) = d_{tables}(q_3.FROM, q_4.FROM) = 0$$

Finally,

$$D(q_1, q_2) = d_{conj}(b_1, b_2) + d_{tables}(q_1.FROM, q_2.FROM) = 0.969;$$
$$D(q_3, q_4) = d_{conj}(b_3, b_4) + d_{tables}(q_3.FROM, q_4.FROM) = 0.96915;$$
$$D(q_1, q_4) = d_{conj}(b_1, b_4) + d_{tables}(q_1.FROM, q_4.FROM) = 0.972.$$

3.3.4.2 Discussion

While the notion of access area is worthy, the distance function has several shortcomings:

(1) The distance function is redundant, as follows: Formula 3.6 sums up the distance of the access tables, calculated using the Jaccard coefficient, as well as the distance of the conjunctions in the filtering conditions. If two queries have the same attributes in the filtering conditions, they have common tables in the FROM clause as well. Taking the distance of the tables accessed when one already calculates a distance of the filtering conditions is redundant. The following example illustrates this.

Example 3.8 Think of a query log containing the queries:

q_1: `SELECT * FROM Cities C WHERE C.latitude BETWEEN 30 and 50`

q_2: `SELECT * FROM Cities C, Countries Cs WHERE C.latitude BETWEEN 40 and 60 AND C.countryId = Cs.id`

The access areas of these queries are:

$q_1 : \sigma_{Cities.latitude \geq 30 \wedge Cities.latitude \leq 50}(Cities);$

$q_2 : \sigma_{Cities.latitude \geq 40 \wedge Cities.latitude \leq 60}(Cities \times Countries).$

The first addend of the distance measure is as follows:

$d_{tables}(q_1.FROM, q_2.FROM) = 1 - |Cities|/|Cities, Countries| = 1/2.$

To calculate $d_{conj}(q_1.WHERE, q_2.WHERE)$, the authors rely on the domain of a column. For attribute latitude in Example 3.8,

$dom(Cities.latitude) = [-90; 90].$

Hence the width of this attribute is:

$width(Cities.latitude) = |90 - (-90)| = 180.$

With predicates in the two queries referring to the same single column,

$d_{conj}(q_1.WHERE, q_2.WHERE) = 1 - (overlap(latitude))/width(latitude) = 1 - 10/180 = 17/18.$

Thus, the overall distance in this example is even more than 1:

$D(q_1, q_2) = 1/2 + 17/18 = 13/9.$

Because two distances are summed up, the result may be an overall distance greater than 1, while this ought to be the value indicating maximally dissimilar queries. Summing up values with different meanings/with different units of measure does not yield results with a clear meaning. One might argue that addends show the degree of dissimilarity – this is their common unit. Then this degree should have at least the same range. However, as Example 3.8 has shown, this is not true: $d_{tables} \in [0;1], d_{conj} \in [0;\infty]$.

(2) The distance of two queries depends on the width of the attributes, see Example 3.8. Hence one cannot come up with a maximum distance in advance. That renders the choice of threshold values for clustering algorithms like DBSCAN difficult.

(3) The similarity function presented in the paper is not a semi-metric. To show this, we calculate the distance of two identical queries from Example 3.8:

$$d_{tables}(q_1.FROM, q_1.FROM) = 0.$$

$$d_{conj}(q_1.WHERE, q_1.WHERE) = 1 - \frac{overlap(latitude)}{width(latitude)} = 1 - \frac{20}{180} = \frac{8}{9}.$$

$D(q_1, q_2) = 0 + \frac{8}{9} = \frac{8}{9}$, while the reflexivity condition requires that $D(q_1, q_1) = 0$.

(4) The distance calculates the overlap of the access areas even if the two queries filter different attributes. The following example illustrates that this may be problematic.

Example 3.9 Think of a query log containing the queries:

q_1: `SELECT * FROM T WHERE a = 1`

q_2: `SELECT * FROM T WHERE b = 2`

In this case, the authors propose to set $d_{conj}(q_1.WHERE, q_2.WHERE)$ to the share of the joint space of the columns involved occupied by $q_1.WHERE$ and $q_2.WHERE$. So these two queries might end up in the same cluster. The clusters then might become too big and consist of disjoint areas of the data space.

In our opinion, these shortcomings impact the identification of user interests based on clusters severely. Namely, when a cluster represents several user interests, one cannot distinguish between them.

3.3.5 Summary

Table 3.1 shows the FB, WB and AAB representations of Query q_1 from Example 2.22. FB is structure-oriented, WB is data-oriented, AAB is somewhere in between, introducing access areas. Since an AAB representation is not a feature vector, one cannot use standard similarity measures, but an AAB similarity function is needed. To get the FB representation, one only needs the query. For WB, in turn, the query and access to the data are needed. AAB does not require the entire data, only some statistical properties, like extreme values of an attribute.

Table 3.1: Query representations of q_1 from Example 3.3

Method	**Query representation**
FB [Kho+10]	$\{f_{*}^{SELECT}, f_{Employees}^{FROM}, f_{Employees.department=<string>}^{WHERE}\}$
WB [CEP09]	{(4352, John, Doe), (4322, Mary, Smith), (4152, Ivan, Green),. . . , (4356, Boris, Johnson), (4322, David, Black)}
AAB [Ngu+15]	$\sigma_{department='Sales'}(Employees)$

Table 3.2: Similarity functions

Sources	**Purpose**	**Similarity function**	**Limitations**
FB [Kho+10] [Eir+14]	SQL queries autocompletion	Depends on the structure of the queries	Considers only the structure of query, not particular values in filtering conditions. Hence, the result of clustering does not reflects users'interests on content level, only meta-data level.
WB [CEP09]	SQL query recommendation	Depends on the tuples in the corresponding result sets	- Scalability due to necessity of re-running SQL statements; - Spoils any further analysis of the query log; - Cannot compare queries which do not return any data; - Might consider queries as similar when their filtering conditions refer to different attributes.
AAB [Ngu+15]	Query clustering of the aim of finding users' interests	Depends on overlap of the filtering conditions of corresponding queries and the width of access for attributes in filtering condition	- Redundancy in the distance function; - The maximum distance is undefinable; - The distance function is not a semi-metric; - Might consider queries as similar when their filtering conditions refer to different attributes.
AAB overlap	Query clustering, association rules mining – SQL query recommendation	Depends on overlap of corresponding filtering conditions	Does not capture closeness of access areas, only overlap.
AAB closeness	Query clustering, association rules mining – SQL query recommendation	Depends on overlap of corresponding filtering conditions	- Works only with ordinal attributes- Sensitive to SWS - "Gravity" effect

3.4 Recommender systems and their Application to SQL QRS

In this section, we first review the techniques of generating recommendations. Then we discuss how to apply them to the SQL context.

3.4.1 "Classical" Recommender Systems (RSs)

Recommender Systems (RSs) are software tools and techniques suggesting items to a consumer [RRS11a]. According to [AT05], the recommendation problem is estimating ratings for items that a consumer has not seen. This does not require consumers to score items explicitly. For instance, if a consumer has bought an item or has viewed it, this may already constitute an assessment. Based on how recommendations are generated, RSs fall into three categories:

(1) Content-based (CB) recommendations [BS97] are based on the consumer's past preferences. They recommend items similar to the ones a consumer has preferred so far.

(2) Collaborative filtering (CF) recommendations [Agg+99] use the preferences of other consumers: They recommend items that people with similar taste have chosen. CF falls into two classes:

   (a) Memory-based methods directly utilize consumer-item ratings stored in the system to predict ratings for new items.
   (b) Model-based approaches use the ratings to learn a model. One is matrix factorization, where vectors of latent factors represent both items and consumers. If the vectors of a consumer and of an item are similar, the item is recommended to the consumer.

(3) Hybrid approaches combine the earlier two categories.

3.4.2 SQL Query Recommendations – Applying Conventional Approaches

We now map the above concepts to the domain of suggesting SQL queries. So conventional recommendation techniques (which have been proven to be valid) become applicable. Since our goal is to recommend SQL queries, items correspond to queries. We now discuss CB and CF in the context of SQL queries.

3.4.2.1 Content-based SQL Query Recommendation

Content-based recommendation means finding similar queries. A query similarity function (QSF) $S(q_1,q_2)$ is required. To recommend full and data-aware queries, $S(q_1,q_2)$ must distinguish between queries with identical structure, but different filtering conditions. Filters may occur in the WHERE or HAVING clause or in the FROM clause if it contains a UDF which requests data from a table based on some condition. Content-based SQL query recommendation suggests queries from a query log which are most similar to a submitted one.

3.4.2.2 Collaborative Filtering in SQL Recommendation

A difference compared to "classical" recommendations when it comes to recommending SQL queries is that the set of items, i.e., all possible SQL requests, may now be infinite. In line with this, the majority of queries occurs only once, in our case study and elsewhere [Sin+07]. So a model-based collaborative approach like matrix factorization is expected to be ineffective here.

To implement a memory-based CF approach, one has to find users (user sessions) with information needs similar to the current one (us_0). This calls for a way to quantify the similarity of user sessions $USS(us_1,us_2)$.

To conclude the section, CB recommendation finds similar queries. CB requires a query similarity function $S(q_1,q_2)$. CF requires a similarity function for user sessions $USS(us_1,us_2)$. Both similarity functions rely on how queries are represented. That is the topic of Section 3.3. We present a more detail analysis of query similarity functions in Chapter 5.

3.4.3 An Overview of SQL QRS

In this section, we review existing methods that generate SQL query recommendations in chronological order. We discuss how they match the requirements from the introduction.

3.4.3.1 Witness-Based QueRIE (WB QueRIE)

SQL query recommendations have been studied for almost a decade. One of the first proposals, [CEP09], is a hybrid technique, combining CB and CF principles. WB QueRIE first builds n vectors each summarizing a user session, $S_1,\ldots,S_n$. It uses the WB representation, both binary and result-based schemes (see Section 3.3.3). For a current user session us_0 the method first builds a so-called *predicted summary*

S_0^{pred}. It is a combination of S_0 itself and of summaries $\{S_1, \ldots, S_n\}$ with similarities $sim(S_0, S_1), \ldots, sim(S_0, S_n)$:

$$S_0^{pred} = \alpha \times S_0 + (1 - \alpha) \times \frac{\sum_{1 \leq i \leq h} sim(S_0, S_i) \cdot S_i}{\sum_{1 \leq i \leq h} sim(S_0, S_i)} \tag{3.10}$$

where h is the number of user sessions a recommendation is based on. Note that S_0^{pred}, a query and a summary of a user session are vectors of length tn, where tn is the number of distinct tuples in a database. The queries from a log most similar to S_0^{pred} (according to binary or result-based similarity) form the recommendations. WB QueRIE has the following limitations:

(1) It has the shortcomings of WB listed in Section 3.3.3.

(2) The similarities between S_0^{pred} and $S_1, \ldots, S_n$ need to be calculated every time the active user submits a new query. That leads to n vector comparisons. The length of each vector is equal to the number of distinct tuples in a database. In the original experiment [CEP09], there were already 13,602,430 witnesses[†], from only 6713 queries. That implies comparing 6713 vectors, each of length 13,602,430. As a result, scalability is poor. The follow-up paper [Eir+14] reports response time of around 5 minutes even for that relatively small log.

(3) The memory required grows with the number of queries and witnesses. If each tuple is an identificator of 4 bytes (Integer), to reproduce an experiment from [CEP09], one must allocate 6713 vectors of length 13,602,430, i.e., $13,602,430 \cdot 6713 \cdot 4 \approx 340$ GB.

(4) The method does not support GROUP BY queries. When a query aggregates the result, it is impossible to "get back" to the individual tuples which have gone into the result.

(5) Hybridization, as in Formula (3.10), merges the results of many queries. It leads to S_0^{pred}, which does not represent a clear user interest, but many of them.

Example 3.10 A current user session us_0 consists of two queries presented in Table 3.3. q_2 of us_0 is an unseen query. us_0 (without q_2) is most similar to user sessions us_1 and us_2. S_0^{pred} includes all tuples which belong to us_0 (till q_1^0), us_1 and us_2. In SQL, S_0^{pred} is the following:

```
SELECT * FROM Cities C
WHERE (C.population BETWEEN 1000 AND 4000) OR C.country = 'USA'
```

But S_0^{pred} is not similar to q_2^0 since they query different tables.

[†]The number of tuples was not specified, but it is not less than the number of witnesses.

Table 3.3: Three user sessions from a query log

User session	Query	SQL statement
us_0	q_1^0	`SELECT * FROM Cities CWHERE C.population BETWEEN 2000 AND 4000`
	q_2^0	`SELECT * FROM Appartments A WHERE C.cityname = 'Madrid'`
us_1	q_1^1	`SELECT * FROM Cities C WHERE C.population BETWEEN 1000 AND 2500`
us_2	q_1^2	`SELECT * FROM Cities C WHERE C.country = 'USA'`

WB QueRIE does hybridization on the level of witnesses (i.e., query representations) "early on". So one cannot distinguish which recommendation comes from CB and which from CF, or even which suggestion comes from which user session. Instead, it combines witnesses of multiple queries in an "unfocused" predicted summary S_0^{pred}, which does not reflect any particular user interest but is a mixture of several ones. Moreover, if an unseen query accesses other tables than the previous SQL requests of a current user session, as in Example 3.10, WB QueRIE is not able to help.

We will investigate whether other approaches to hybridization will yield better results.

3.4.3.2 FlexRecs

FlexRecs [KBGM09] supports recommendations as SQL statements over structured data. As WB QueRIE, it requires access to the database. Queries are compared based on their result sets. Moreover, it requires execution plans of the SQL requests, which is often not possible to collect. Scalability-wise, FlexRecs is intended for relatively small data (around 10000 entities in a table).

3.4.3.3 Recommending JOIN Queries

YPS[‡] [YPS09] assists in writing queries with several joins. It builds a graph out of a query log where nodes are tables and edges are join constraints. According to the classification in Section 3.3, a query is represented as a set of features (FB): tables, join conditions. In the experiments, YPS gets the tables which the query has requested. The approach recommends join constraints. While the method is scalable, it does not recommend full and data-aware queries.

[‡]The approach does not have a name in the original paper, so we use the initials of the authors. The same with AGGMR.

3.4.3.4 SnipSuggest

SnipSuggest [Kho+10] recommends auto-completion of a partly written query, based on the query log. Here, the FB query is tables, views, UDFs, attributes in the SELECT clause, predicates in the WHERE clause, GROUP BY attributes. The recommendation engine of SnipSuggest relies on a directed acyclic graph (DAG), built from the log. Sequences of features form vertexes of the graph. Let us take $s\phi_1$ as a sequence of features and $s\phi_2$ as another one built from $s\phi_1$ by appending feature $\phi_{1.i}$. The weight of an edge between $s\phi_1$ and $s\phi_2$ is the probability of $\phi_{1.i}$ following $s\phi_1$. SnipSuggest transforms the user's partially written query into a sequence of features $s\phi_j$, which is mapped onto a path in the DAG. It recommends features $\{\phi_{1.j}, \dots \phi_{k.j}\}$, which complement $s\phi_j$ to form $(s\phi_1, \dots, s\phi_k)$. The ranking of features depends on the weight of the edges.

Overall, SnipSuggest has developed the earlier idea of utilizing a graph from YPS [YPS09]. As YPS, SnipSuggest is fast and scalable. However, it does not recommend the entire query. Nevertheless, the corresponding feature-based query representation and the idea of using DAG seem promising. We will adapt them for our purposes.

3.4.3.5 Fragment-based QueRIE (FB QueRIE)

To cope with the poor scalability of WB QueRIE [CEP09], the authors propose a FB QueRIE approach in [Eir+14]. It, however, does not recommend data-aware queries, only structures of queries. A *fragment* is what called a feature in [Kho+10]. Thus, from now on we use the term *feature* instead of *fragment*. Similarly to [CEP09], [Eir+14] first builds n vectors summarizing user sessions $S_1, \dots, S_n$. Each coordinate of a vector represents the presence of some feature in the session. The authors utilize either binary or weighted scheme (see Section 3.3.2).

As the second step, [Eir+14] computes pairwise feature similarities $sim(\rho, \psi)$ offline. For two features ρ and ψ, one gets $sim(\rho, \psi)$ by comparing vectors $F_\rho = (w_1^\rho, \dots, w_n^\rho)$ and $F_\psi = (w_1^\psi, \dots, w_n^\psi)$ of the presence of features in all user sessions$\{S_1, \dots, S_n\}$. $sim(\rho, \psi)$ reflects how often two features appear together in the queries of a log. To recommend a query for a current user session us_0, the algorithm builds a predicted summary S_0^{pred}. The approach recommends queries with summaries most similar to S_0^{pred}.

As with WB QueRIE, FB QueRIE results in an S_0^{pred}, which includes too many features. Therefore, we doubt that recommending query structures as in FB QueRIE is the best way. We will investigate on this point when proposing our suggestion of query templates and compare it to FB QueRIE.

Table 3.4: Compliance with the requirements

Property / Approach	Requirements			QR	CB/CF/ Hybrid	Is applicable
	1	2	3			
WB QueRIE [CEP09]	+	+	-	WB	Hybrid	Yes. However, the input data needs sampling due to scalcbility issue.
FlexRecs [KBGM09]	+	+	-	WB	-	No. It demands the access to the database to see query plans.
YPS [YPS09]	-	-	+	FB	CF	No. Does not provide full and data aware queries.
SnippSuggest [Kho+10]	-	+	+	FB	CF	No. Does not provide full and data aware queries.
FB QueRIE [Eir+14]	-	+	+	FB	Hybrid	No. Does not provide data aware queries.
AGGMR [Ali+15]	-	+	+	FB	CF	No. Does not supprt range queries.

FB QueRIE [Eir+14] claims it to be much faster than WB QueRIE [CEP09]. However, it is more complex than other FB methods: To predict a single current user session, one still has to compare S_0^{pred} with all the queries from a log.

3.4.3.6 Recommending OLAP Sessions

[Ali+15] focuses on recommendations in OLAP databases. While the query language used for OLAP queries may be different from SQL, we review it here, to borrow a notion from this approach. The authors aim at the common OLAP interactions (drill-down, slice-and-dice, etc.). They conclude that one should focus on sessions, not on individual queries. A query representation is a mixture of the FB and AAB representations: It includes attributes in SELECT and GROUP BY clauses as well as filtering conditions. The principal limitation is that a query may only have equality comparisons. Range queries are not allowed.

The AGGMR approach calculates the similarity of OLAP sessions based on query similarity function $\sigma_{que} \in [0, 1]$, described in [Ali+14]. It applies collaborative filtering (CF). The approach works in three steps: (1) It searches the log for similar user sessions. (2) It extracts the most relevant query sequences (subsessions) within similar user sessions. (3) It adapts (fits) the top-ranked subsession to the current session of the user.

Finding similar sessions is order sensitive: It utilizes the Smith-Waterman algorithm [SW+81]. In the fitting step, the authors modify the recommended query depending on the differences of the current user session and a similar one.

AGGMR features several exciting ideas, namely (1) order-sensitive similarity of user sessions and (2) query modification. However, the original method cannot work with range queries.

3.4.4 Resume

Table 3.4 summarizes the approaches and their compliance with the requirements listed in the introduction. No existing approach satisfies all requirements: They either

(1) do not produce data-aware queries or

(2) do not support range queries or

(3) require access to query plans.

We have found WB QueRIE to be conditionally appropriate: If the query log is small, it fulfills the requirements. WB QueRIE suggests only queries from a query log. If a user wants to explore data that nobody has requested before, the recommendation will not be good.

4 Cleaning Antipatterns in an SQL Query Log

In this chapter, we present an approach to clean SQL query log by removing or solving antipatterns from it. This is a general step for any further SQL query log analysis: clustering (Chapter 5) and query recommendation (Chapter 6). We reused our corresponding publication [ASB18], including figures and algorithms.

Let us clarify what we meant by removing and solving.

Example 4.1 Get back to a query log from Table 1.1. If antipatterns are *solved*, the output log would look like in Table 4.1. If antipatterns are *removed*, the log would be like in Table 4.2.

Thus, when *solving*, one gets the queries, which return the same data, as an original log. The only difference is that a query log does not contain antipatterns any more. When one *removes* antipatterns from a query log, some information will be deleted.

We first provide formal definitions of a pattern and antipatterns for our context. We also construct solutions for solving antipatterns whenever possible. Then we present implementation details and results.

4.1 Patterns and antipatterns: formal definitions, detection rules and solving solutions

Antipatterns are patterns with negative effects. That is why the notion of a pattern is essential in the context of antipattern detection. This section formally defines and introduces the properties of a pattern. We also need precise definitions of the Stifle

Table 4.1: A series of statements from one user

#	Statements	Result
1	**SELECT** E.empId **FROM** Employees E **WHERE** E.department = ‘sales’	12
2	**SELECT** E.empId, E.name, E.surname, 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	John, Doe, 03/12/1985 01259863448, 36

Table 4.2: A series of statements from one user

#	Statements	Result
1	**SELECT** E.empId **FROM** Employees E **WHERE** E.department = 'sales'	12

and CTH antipatterns to facilitate their discovery and automatic solving. This section contains these definitions.

4.1.1 Definitions for a Database Pattern

We identify a pattern as a sequence of queries which represents certain functionality. Starting with the definition of patterns from software engineering, we first describe the properties of patterns in databases informally. This discussion then leads to a formal definition of a pattern. We then use it throughout this thesis.

4.1.1.1 Demonstrative example

In software engineering, a pattern is a recurring solution schema to a standard problem deployed in a certain context [Dud+03]. In this thesis, we redefine a pattern as a sequence of SQL queries in a query log. In a nutshell, wee see a pattern as it shows itself in a query log.

Example 4.2 Think of the database of a shoe retailer. Buying a pair of shoe 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 IS 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 a short time between them;
(3) have a rather frequent occurrence;
(4) be from the same user.

4.1.1.2 Definition of a pattern

What we have learned from Example 4.2, is that a pattern manifests itself in a query log as a sequence of queries. Values in filtering conditions or parameters of DML requests are ignored. Let us first informally define the notion of a *template* as an SQL query, where each value is replaced with a placeholder. Now we can formally define a pattern:

Definition 4.3 A pattern p is a sequence of query templates $(t_1, \dots, t_n)$.

The notion of a *template* requires additional clarification. Let us consider the following example.

Example 4.4 Queries q_1 and q_2 listed below are from different users:

q_1:
```
SELECT MODEL, SIZE into curr_model, curr_size
     FROM BarCodesInfo WHERE ID = 12;
```

q_2:
```
SELECT SIZE, MODEL into curr_model, curr_size
     FROM BarCodesInfo WHERE ID = 13;
```

Seeing a template as a string, where values replaced by placeholders, these two queries belong to different templates $t_1 = template(q_1)$ and $t_2 = template(q_2)$:

t_1:
```
SELECT MODEL, SIZE into curr_model, curr_size
     FROM BarCodesInfo WHERE ID = <num>;
```

t_2:
```
SELECT SIZE, MODEL into curr_model, curr_size
     FROM BarCodesInfo WHERE ID = <num>;
```

However, both templates, t_1 and t_2, implement the same functionality - obtain model and size by a barcode.

Example 4.4 shows that one needs more flexibility when defining a template. In a nutshell, an order of attributes, tables, or filtering conditions should be ignored. We propose to see a *template* as a *FB query* from [Eir+14]. To keep things simple, we will utilize the notation from [Kho+10] for a feature. Thus, it could also be seen as a query representation from [Kho+10] with the only difference: it is now not a sequence of features, but a set.

Definition 4.5 A template $t = template(q)$ of a query q is a set of features of q, where a feature is defined as in [Kho+10].

Thus, two queries from Example 4.4 belong now to the same template:

$$template(q_1) = template(q_2) = \\ \{f^{SELECT}_{model}, f^{SELECT}_{size}, f^{FROM}_{BarCodesInfo}, f^{WHERE}_{BarCodesInfo.Id=<num>}\}.$$

Let us define a specific sub-classes of features.

Definition 4.6 $template(q)^{SELECT}$ are all features, which start from f^{SELECT}, i.e. ones in SELECT clause.

Definition 4.7 $template(q)^{FROM}$ as all features, which start from f^{FROM}.

Definition 4.8 $template(q)^{WHERE}$ as all features, which start from f^{WHERE}.

Definition 4.9 $template(q)^{GROUPBY}$ as all features, which start from $f^{GROUPBY}$.

Going back to Query q_1 from Example 4.4,

$$template(q_1)^{SELECT} = t_1^{SELECT} = \{f^{SELECT}_{model}, f^{SELECT}_{size}\}, \\ template(q_1)^{FROM} = t_1^{FROM} = \{f^{FROM}_{BarCodesInfo}\}, \\ template(q_1)^{WHERE} = t_1^{WHERE} = \{f^{WHERE}_{BarCodesInfo.Id=<num>}\}$$

4.1.1.3 Properties of a Pattern

Let us introduce a few definitions to find an instance of a pattern in a query log.

Definition 4.10 A user $u = user(q)$ of a query q is a user, who has run q.

Definition 4.11 A moment in time $time(q)$ is the time, when q was requested.

Definition 4.12 An instance $(q_1, \dots, q_n)$ of a pattern $p = (t_1, \dots, t_n)$ is a sequence of queries in the query log such that

(1) $\forall i, 1 \le i \le n, t_i = template(q_i)$
(2) $user(q_1) = user(q_2) = \dots = user(q_n)$

(3) $time(q_1) \leq time(q_2) \leq \cdots \leq time(q_n)$
(4) $\forall i, 1 \leq i \leq n, \nexists q_x \notin (q_1, \ldots, q_n) \quad where \quad user(q_x) = user(q_i) \wedge time(q_i) \leq time(q_x) \leq time(q_{i+1})$

The last axiom states that there are no other requests from the same user within time window $time(q_1); time(q_n)$.

Definition 4.13 The *frequency* of a pattern in a log is the number of its *instances* occurring in the log.

Definition 4.14 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.

4.1.2 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 4.1.1. For each selected antipattern, we provide a detection rule. If an antipattern has a solving solution, we consider it *solvable*.

4.1.2.1 The Stifle Antipattern

Our literature review (see Section 3.2.1) has yielded the following specific characteristics of the Stifle:

(1) Small average number of result rows;
(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 4.15 A *Stifle* antipattern is a pattern $p = (t_1, \ldots, t_n)$ such that

(1) $|Ints(q_1)| = |Ints(q_2)| = \cdots = |Ints(q_n)| = 1$
(2) $operators(q_1)[1] = \cdots = operators(q_n)[n] =' ='$
(3) $Ints(q_1), \ldots, Ints(q_n)$ are key attributes

Note that Definition 4.15 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 positive detected antipatterns. Our solving scheme for *Stifle* 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.

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 4.16 The following is a DW-Stifle antipattern:

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

The formal definition is as follows:

Definition 4.17 A *DW-Stifle* is a Stifle (see Definition 4.15) $p = (t_1, \ldots, t_n)$ such that

(1) $t_1^{SELECT} = t_2^{SELECT} = \cdots = t_n^{SELECT}$
(2) $t_1^{FROM} = t_2^{FROM} = \cdots = t_n^{FROM}$
(3) $t_1^{WHERE} = t_2^{WHERE} = \cdots = t_n^{WHERE}$
(4) $P(q_1)[1] \neq P(q_2)[1] \neq \cdots = P(q_n)[1]$

See Definitions 4.6, 4.7, 4.8 and 2.16 for details.

The fourth condition in Definition 4.17 requires a comparison of predicates. This task is difficult to comply in general. As we saw in the literature review, AAB query similarity (Section 3.3.4), which compares the predicates of two queries, is quite complex and has some shortcomings. Luckily for us, as we deal with Stifle, queries have only one attribute with equality operation in their filtering condition. Those predicates are easy to compare: *iff* the values are equal, two predicates are equal. Otherwise, they are not alike.

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

Example 4.18 The solving solution for Example 4.16 is :

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

Compared to the solving solution in Example 3.2 we now get one SQL statement instead of several ones.

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 4.19 A DS-Stifle instance is as follows:

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

Definition 4.20 A *DS-Stifle* is a Stifle $p = (t_1, \ldots, t_n)$ with the following characteristics:

(1) $t_1^{SELECT} \neq t_2^{SELECT} \neq \cdots = t_n^{SELECT}$
(2) $t_1^{FROM} = t_2^{FROM} = \cdots = t_n^{FROM}$
(3) $t_1^{WHERE} = t_2^{WHERE} = \cdots = t_n^{WHERE}$
(4) $P(q_1)[1] = P(q_2)[1] = \cdots = P(q_n)[1]$

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

Example 4.21 The solving solution for Example 4.19 is :

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

DF-Stifle

Stifels with different FROM statements are named *DF-Stifle* ('different FROM' Stifle).

Example 4.22 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;
```

The formal definition is as follows:

Definition 4.23 A *DF-Stifle* is a Stifle $p = (t_1, \ldots, t_n)$ where

(1) $t_1^{SELECT} = t_2^{SELECT} = \cdots = t_n^{SELECT}$

(2) $t_1^{FROM} \neq t_2^{FROM} \neq \dots = t_n^{FROM}$
(3) $t_1^{WHERE} = t_2^{WHERE} = \dots = t_n^{WHERE}$
(4) $P(q_1)[1] = P(q_2)[1] = \dots = P(q_n)[1]$

Example 4.24 illustrates our solving scheme:

Example 4.24

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

4.1.2.2 The Circuitous Treasure Hunt Antipattern

The distinctive feature of the *CTH* (The Circuitous Treasure Hunt) antipattern is dependency of the queries. As re-querying is not feasible (see Section 1.1) 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 follow-up query/queries.

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 4.25 A *CTH candidate* is a pattern $p = (t_1, \dots, t_n)$ such that:

(1) $t1 \neq t2$
(2) $\exists\ intst,$ where $\forall i \in [2 \dots n]\ (f_{intst}^{SELECT} \in t_1^{SELECT}) \wedge (intst \in Ints(q_i))$

Here *intst* of the second condition is an attribute, which appears in both, SELECT clause of q_1 and WHERE clause of queries $q_2, \dots q_n$.

Our respective detection method looks for patterns which satisfy Definition 4.25. 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.

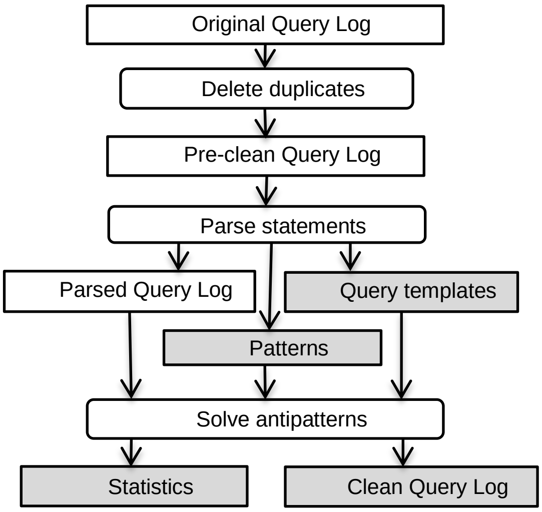

Figure 4.1: Processing Steps

4.2 Implementation of Solving Antipatterns in an SQL Query Log

In this section, we describe the realization of our approach. We first give an overview of the architecture of our framework that solves antipatterns. We then introduce the components of the processing pipeline in more detail. There is a web page* of our framework 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. Figure 4.1 shows the respective workflow. Rectangular boxes stand for input data, rounded boxes for processing steps, and gray boxes for results. There are several outcomes extracted from a SQL log. In contrast, the log is the only input.

To make it more understandable and fascinating, we put examples of a query log and how it is transformed all the steps of the way.

*https://dbis.ipd.kit.edu/2500.php

4.2.1 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. Table 4.3 list an example of an original query log.

4.2.2 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. That is because a large number of them may indicate a refactoring of a particular application (logging system, at the best case).

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 us to find most of the duplicates. Section 4.3.2 will discuss respective empirical results.

After deleting duplicates an example query log from Table 4.3 became a query log from Table 4.4.

4.2.3 Parsing Statements and Parsed Query Log

After deduplication, there may still be syntactically incorrect query statements. In this step, we parse all queries into templates and store filtering conditions for all of them. If the parsing finds a syntax error, the process will not consider the statement any further. We also exclude non-select statements.

Table 4.7 contains an example of a parsed log, which is made from the log without duplicates (see table 4.4). Each query of a parsed log also contains the link to a pattern (from the Patterns data table) and a query template (from the Query templates data table) a statement belongs to. The list of templates for the log in 4.7 is in Table 4.5, the list of patterns is in table 4.6. If a parsed statement satisfies the definition of an antipattern (Definition 4.15 to Definition 4.25), it is marked as an antipattern of the respective type.

*"d" stays for "department". It is shortened due to space limit.

Table 4.3: Original query log

#	Timestamp	Statements
1	12/04/2017 09:35:40	`SELECT E.Id FROM Employees E WHERE E.department = 'sales'`
2	12/04/2017 09:36:30	`SELECT E.Id FROM Employees E WHERE E.department = 'sales'`
3	12/04/2017 09:37:10	`SELECT E.name, E.surname FROM Employees E WHERE E.id = 12`
4	12/04/2017 09:37:11	`SELECT E.name, E.surname FROM Employees E WHERE E.id = 12`
5	12/04/2017 09:37:40	`SELECT E.name, E.surname FROM Employees E WHERE E.id = 15`
6	12/04/2017 09:38:05	`SELECT E.name, E.surname` `FROM Employees E WHERE E.id = 16`

Table 4.4: Query log without duplicates

#	Timestamp	Statements
1	12/04/2017 09:35:40	`SELECT E.Id FROM Employees E WHERE E.department = 'sales'`
2	12/04/2017 09:36:30	`SELECT E.Id FROM Employees E WHERE E.department = 'sales'`
3	12/04/2017 09:37:10	`SELECT E.name, E.surname FROM Employees E WHERE E.id = 12`
5	12/04/2017 09:37:40	`SELECT E.name, E.surname FROM Employees E WHERE E.id = 15`
6	12/04/2017 09:38:05	`SELECT E.name, E.surname FROM Employees E WHERE E.id = 16`

Table 4.5: Table of templates

Template Id	Template
1	$\{f^{SELECT}_{id}, f^{FROM}_{Employees}, f^{WHERE}_{department=<string>}\}$
2	$\{f^{SELECT}_{name}, f^{SELECT}_{surname}, f^{FROM}_{Employees}, f^{WHERE}_{id=<num>}\}$

Table 4.6: Table of patterns

Pattern Id	Pattern (series of Template Ids)
1	(1, (2))
2	(2, (2))

Table 4.7: Parsed query log

#	Time-stamp	Statements	Filtering conditions	Temp;ate Id	Pattern Id	Type of anti-pattern
1	2/4/2017 09:35:40	`SELECT FROM Employees WHERE department ='sales'`		Syntax error: not processed		
2	2/4/2017 09:36:30	`SELECT E.Id FROM Employees WHERE department ='sales'`	Employees. department = 'sales'	1	1	CTH candidate
3	2/4/2017 09:37:10	`SELECT name, surname FROM Employees WHERE id = 12`	Employees. id = 12	2	1, 2	CTH candidate, DW-Stifle
5	2/4/2017 09:37:40	`SELECT name, surname FROM Employees WHERE id = 15`	Employees. id = 15	2	1, 2	CTH candidate, DW-Stifle
6	2/4/2017 09:38:05	`SELECT name, surname FROM Employees WHERE id = 16`	Employees. id = 16	2	1, 2	CTH candidate, DW-Stifle

Table 4.8: A clean query log

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

4.2.4 Query Templates and Patterns

We compute statistics for each template and pattern using the *frequency* and *userPopularity* properties (Definition 4.13 and Definition 4.14, see Section 4.1.1.3). 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 [KA10]. An example of it is as follows:

Example 4.26
```
SELECT * FROM Bugs WHERE assignedto = NULL
SELECT * FROM Bugs WHERE assignedto <> NULL
```

Since neither equality nor inequality returns 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:

Example 4.27
```
SELECT * FROM Bugs WHERE assignedto IS NULL
SELECT * FROM Bugs WHERE assignedto IS NOT NULL
```

We now provide a formal definition of SNC.

Definition 4.28 A SNC is a pattern $p = (t_1)$ where

(1) $P(q_1)$ consists of "NULL'
(2) $\exists i$, where $operators(q_1)[i] = '=' $ or $operators(q_1)[i] = '\neq'$

The solving solution is rather straightforward: 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 4.28 in the parse step, as we have done with antipatterns described in Section 4.1. Since there is a solving solution as well, one can include it in the step "Solve antipatterns". From now on, each SNC detected would be solved.

4.2.5 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 4.19 or Example 4.18). The procedure returns a cleaned query log and statistical information regarding antipatterns solved. We identify how many of them we have encountered in the query log, how many we have solved. The following is an example of this procedure.

Example 4.29 Table 4.7 contains a query log after parsing. Queries 3, 5 and 6 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 4.8.

In the example, Queries 3, 5 and 6 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 specific 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 solving 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.

4.3 A case Study With SkyServer

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

(1) Answer how many patterns and antipatterns a large real-world log contains;
(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) Showcase the influence of cleaning antipatterns on subsequent analyses.

4.3.1 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 these 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.

4.3.2 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 4.9). We already identify most duplicates when using 1 second as the threshold. That means that duplicates indeed are requests not intended by the user, as we have hypothesized in Section 4.2.2. Most duplicate queries are results from web-form reloads or from errors in applications.

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 behaviour may differ between databases, each SQL log analysis may require its threshold value. Tests similar to the one just described should allow determining this value.

Table 4.9: 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

4.3.3 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 4.10). Overall, cleaning a log with our approach has resulted in 27.5% size reduction. That 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 4.3.6). The instances of the antipatterns cover about 7.5 million statements.

Another benefit due to this size reduction is a decreased load for subsequent downstream analyses, see Section 4.3.8.

4.3.4 Effects of SQL Log Cleaning

To evaluate the effectiveness of solving antipatterns, we compare the most popular patterns before and after running the solving procedure. Figure 4.2 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 4.11 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 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 4.11 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. On the contrary, 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.

4.3.5 Interpretation of Patterns

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

4.3.5.1 What do patterns do?

Table 4.12 contains the most popular patterns in the query log after removing the antipatterns. All five patterns perform a 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 frequency has rank 2, etc.

4.3.5.2 Sliding window search (SWS) pattern

According to Figure 4.3, 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 a sliding window search pattern (*SWS pattern*). We do not classify the SWS pattern as an antipattern since it uses a database in the right way. Our explanation of 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. That is because patterns of medium frequency which come from more users will be labeled as SWS. Table 4.13 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* thresholds. The numbers are in line with our explanation.

The discovery of SWS patterns is essential 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

Table 4.10: 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

Table 4.11: The most popular antipatterns

#	**Frequency**	**Type**	**First template**	**Second template**	**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 WHEREobjid=<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

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 hypothesize that SWS patterns also bog down the prediction quality of association-rule mining. Suppose that we want to suggest the next query based on the previous ones a user has typed so far. If the learning set contains SWS queries, a query recommendation system would suggest it. However, this kind of behaviour (sliding window search) is a "machine download", which does not require query-recommendation assistance. Humans, on the other hand, would benefit from query suggestion without SWS in a recommendation set [Sin+07].

Table 4.12: The most popular patterns

#	Frequency	Templates	Description	Distinct IPs
1	3,349,709	`SELECT g.objid,... FROM photoobjall as g JOIN fgetnearbyobjeq as gn on g.objid=gn.objid left outer join specobj s as gn s.bestobjid=gn.objid`	Gets objects within @r arcmins of an Equatorial point (@ra,@dec)	1
2	3,082,742	`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	`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	`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	`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. It is also some sort of special search.	1

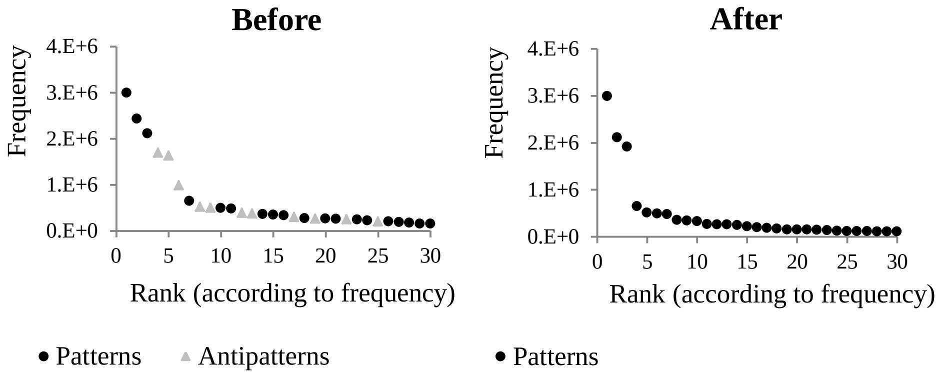

Figure 4.2: The most popular patterns before and after cleaning the log

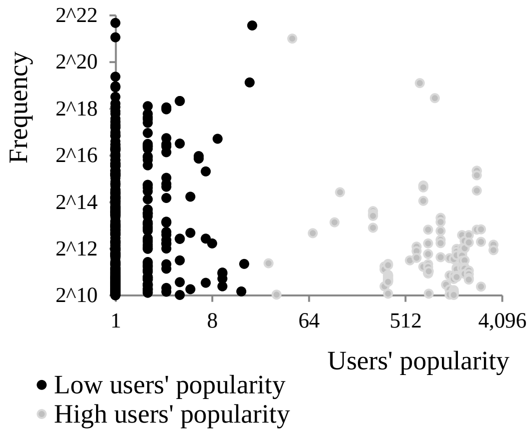

Figure 4.3: Frequency and User Popularity of the Patterns

Table 4.13: 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%

4.3.6 CTH Detection

With our solution, we discover 50 candidates for CTH antipatterns. As mentioned, the decision whether such a pattern is, indeed, 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 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 4.30 Consider an instance of two CTH candidates:

The instances of CTH Candidate 1 comply with Sky-Server Web interface functionality: First, a user looks for all tables in a database, and then chooses table ’Galaxy’ and query it. The time difference between the two queries indicates that the second query has not been issued before the first one – the user has reflected for a while which data he wants to enrich next.

Table 4.14: 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

Table 4.15: 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 11.25.00 AM

The queries in CTH Candidate 2, in contrast, run directly one after another, with no time difference. The first query 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 need further look. 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.

Figure 4.4 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.

4.3.7 Feedback from Domain Experts

To assess the usefulness of our results, we have conducted a study 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

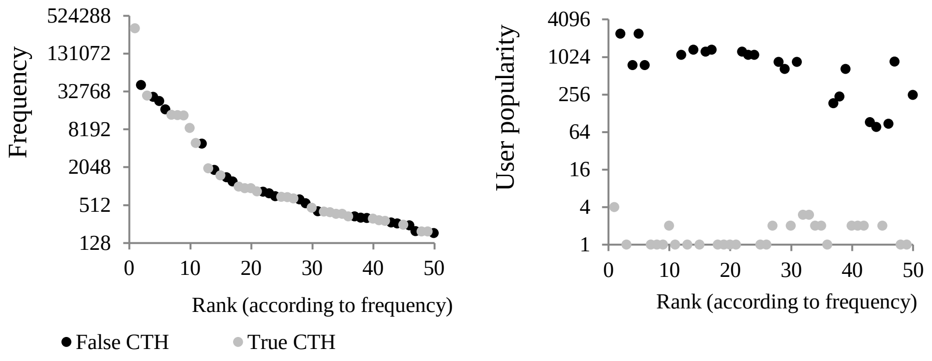

Figure 4.4: Possible and real CTH antipatterns

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 4.11) with a previous query and asks for more data. That 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.

4.3.8 Effects on Downstream Analysis

Even though it is not the core topic of this chapter, 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 [Ngu+15]. It detects user interests via clustering from 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);
(2) *Removal* query log, obtained from the raw query log by removing antipatterns (0.89 million queries);

[†]In the original work of [Ngu+15] a distance has value ranges $[0;\dot{\infty}]$. We have normalized the metric from 0 to 1 to be able to set meaningful thresholds.

(3) *Solved* query log, obtained from the raw query log by solving antipatterns (1 million queries).

In the case of cleaning, we do not delete antipatterns from a log, but rewrite them, as discussed in Section 4.1.2. Hence, the removal log is smaller than the solved one.

It is important to point out that the purpose of this experiment is "only" to show the effect on meta-level: (1) how much clusters one get with raw, solved and removed log, (2) how big they are, (3) how much time does it take to perform clustering. We do not investigate how good in terms of *precision* and *interpretability* (see Section 1.2) the clusters are. As our literature review has discovered, there are certain problems with a similarity function, [Ngu+15] has utilized. We will redefine it and perform an extensive clustering experiment later on in Chapter 5. Therefore this initial experiment serves to indicate if the noise introduced by antipatterns influences subsequent analysis (clustering) in some way.

Figure 4.5 shows the results. Varying the threshold value (from 0.1 to 0.9) has little impact on the number of clusters. That 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. That 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 0.9).

We have found all clusters from the removal log in the raw log. They also present in the cleaned log. That indicates that removing antipatterns indeed is a way to get rid of the 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 Figure 4.5 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 in line with our prediction, i.e., 1.2. Figure 4.6 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';`

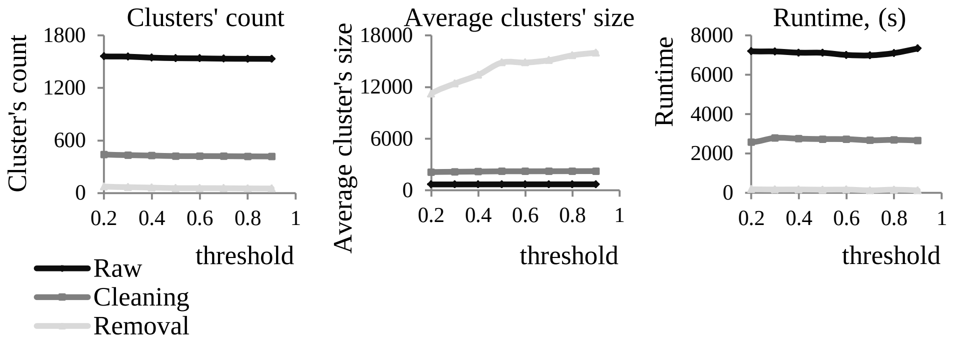

Figure 4.5: Results of an experiment on query clustering

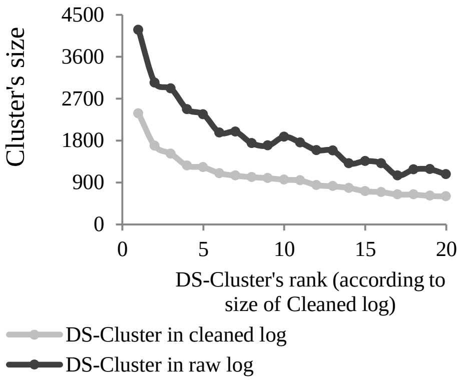

Figure 4.6: DS-Clusters' sizes for cleaned and raw log. Threshold = 0.9

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.

4.4 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 to reveal such information. Antipatterns, however, might falsify such analyses; discovering antipatterns in the query log is beneficial for refactoring and post-processing. To our knowledge, finding database antipatterns in SQL query logs have not been studied before systematically. In this chapter, we have proposed a solution for the detection of patterns and solving antipatterns in a query log. To this end, we have formalized the notion of a 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 behaviour, as a case study on the SkyServer query log has demonstrated.

All in all, our approach is capable of detecting and classifying patterns in a query log. The results show a significant number of instances of antipatterns within the SQL query 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.

5 SQL-Query-Similarity Measures

This current chapter studies the similarity of SQL queries through clustering of SQL query logs to identify user interests *within a data space*. It reuses our corresponding publication [Arz+19]. In a nutshell, after performing a clustering, we want to know not only *how* users work with a database (i.e., which patterns (Definition 4.3) do they utilize), but *what exactly* they are looking for (i.e., which data they are interested in). To this end, we need a data-aware query similarity function.

There are two data aware approaches we have found in the literature: *witness-based (WB)* (Section 3.3.3) and *access-area-based (AAB)* (Section 3.3.4). While WB sees a query as its result set, AAB query is a relational algebra expression. Whereas WB query similarity is standard (Jaccard, cosine) and does not require additional care, current AAB similarity has shortcomings, which make it inapplicable for clustering (see Section 3.3.4.1). Therefore, a new AAB similarity is needed.

5.1 Our AAB similarity functions

Representing a query as its access area captures key details of an SQL request and does not consider the current state of a database, in contrast to the WB approach. However, as we have pointed out in Section 3.3, a different query-similarity function is necessary. We now propose new functions which are still based on the notion of access areas but do not suffer from the shortcomings discussed in Section 3.3.4.1.

5.1.1 Requirements to an SQL query similarity function

Let us first define a query similarity formally:

Definition 5.1 The similarity $S(q_1, q_2)$ of two queries is a function returning a value in $[0;1]$.

$$S(q_1, q_2) = \begin{cases} 1 \text{ if } q_1 = q_2 \\ [0;1) \text{ otherwise} \end{cases} \tag{5.1}$$

It holds that $S(q_1,q_2) = 1 - D(q_1,q_2)$. In general there is no restriction on the range of values of $D(q_1,q_2)$, i.e., $D \in [0;\infty)$. This requirement is there exclusively for our query-distance function. We have introduced it to set a meaningful threshold for the algorithms, which imply knowledge regarding extreme values of D (like DBSCAN).

Certain clustering algorithms impose conditions regarding the distance function used. So-called semi-metric distances work with a wide variety of clustering algorithms. According to [AM+09], such a distance $D(x_i,x_j)$ must satisfy the following conditions on a data set X:

(1) *Symmetry.* $D(x_i,x_j) = D(x_j,x_i)$;
(2) *Positivity.* $D(x_i,x_j) \geq 0$ for all x_i and x_j in X;
(3) *Reflexivity.* $D(x_i,x_j) = 0$ iff $x_i = x_j$;

We will check these conditions when introducing our query-distance measures.

5.1.2 Requirements to an SQL query

InChapter 2 we have defined an SQL query as an SPJ (Select-Project-Join) one, cf. Definition 2.5. When it comes to AAB query similarity and getting adequate clustering results, we have to exclude two more types of queries out of consideration: (1) queries with arithmetic operations in predicates and (2) queries without a filtering condition. Let us discuss these queries justifying our choice:

(1) *Queries with arithmetic operations in predicates.* These queries are complicated in general, for instance, for DBMS query optimizers. The current version of our AAB approach does not cover these queries. The reason is the same as with DBMS optimizers: there is no way knowing in advance which data one gets after applying arithmetic operations. However, to include this type of queries into consideration, one could resort to the WB approach. That is the case particularly since one of our similarity functions, which calculates the overlap of access areas (dubbed AABovl in the following), and WB yield similar results, as we will show in Section 5.2.
(2) *Queries without a filtering condition.* This kind of query is often used in combination with a TOP-n clause. These requests are usually the first ones a user might issue, intending to test the database. In this case, they have relatively little to do with user interests. They also blur the aggregated access area of a cluster they belong to: the cluster spread to the whole domain, bringing no insights of a particular dataspace the users are interested in. In our case study with SkyServer, only 2.7 % of the queries are of this kind. So we have consciously decided to leave them aside in this current study.

5.1.3 Definitions

Definition 5.2 The similarity measure $S(q_1,q_2).a$ of an attribute a of two queries q_1, q_2 is a similarity measure of queries q_1 and q_2 which is defined if the queries both have at least one filtering condition with Attribute a and is undefined otherwise.

For a query pair q_1 and q_2, there can be one or several conditions on Attribute a.

Definition 5.3 The distance $D(q_1,q_2).a$ of two queries q_1, q_2 with respect of Attribute a is the corresponding distance, which is calculated as follows: $D(q_1,q_2).a = 1 - S(q_1,q_2).a$.

Definition 5.4 An ordinal attribute (OA) is one whose values have a natural order.

The values of such an attribute may or may not be from a domain (Definition 2.6) that is continuous.

Definition 5.5 A nominal attribute (NA) is one whose values do not have a natural order.

Definition 5.6 The common interest $comInts(q_1,q_2)$=$Ints(q_1) \cap Ints(q_2)$ of two queries q_1,q_2 is the set of *interests* which occur in both queries q_1 and q_2.

Definition 2.10 features the respective term of an *interest*.

Definition 5.7 The exclusive interest $exclInts(q_1,q_2) = Ints(q_1) \cup Ints(q_2)$ $\setminus comInts(q_1,q_2)$ of two queries q_1, q_2 is the set of interests which occur in only one query.

In line with Definition 2.16, we use the notation $P_i[a]$ for a filtering condition occurring within Query q_i and referring to Attribute a.

Definition 5.8 A set of intervals $A_i^{OA} = \{a_{i.1}^{OA}, \ldots, a_{i.k}^{OA}\}$ of a query q_i is one formed by filtering condition P with ordinal attribute (OA) a.

Example 5.9 Consider Queries q_1 and q_2:

q_1: `SELECT * FROM Cities C WHERE C.latitude`
`BETWEEN 10 and 20 OR C.latitude BETWEEN 40 and 50`

q_2: `SELECT * FROM Cities C WHERE C.latitude NOT BETWEEN 40 and 50`

Query q_1 has the following set of intervals for ordinal attribute *Cities.latitude*:

$$Cities.latitude_1^{OA} = \{[10;20],[40;50]\}.$$

For Query q_2:

$Cities.latitude_2^{OA} = \{[-90;40],[50;90]\}$, because $dom(Cities.latitude) = [-90;90]$ (see Definition 2.6).

We now define how to extract intervals out of a query.

Definition 5.10 The set of intervals $A_i^{OA} = \{a_{i.1}^{OA}, \dots, a_{i.k}^{OA}\}$ of a filtering condition $P_i[a]$ associated with an *ordinal* Attribute a is as follows. If $P_i[a]$ is:

(1) Atomic clause: A_i^{OA} is a singleton set containing exactly the clause;
(2) Conjunction clause $(C_1 \wedge C_2) : A_i^{OA} = C_1 \cap C_2$; it is the intersection of the intervals in C_1 and C_2;
(3) Disjunction clause $(C_1 \vee C_2) : A_i^{OA} = C_1 \cup C_2$; it is the union of intervals in C_1 and C_2;
(4) Negative clause (NOT C): $A_i^{OA} = \neg C$; it is the inverse interval of C.

Definition 5.11 The width $width(a_{i.k})$ of interval $a_{i.k}^{OA}$ is defined as $width(a_{i.k}) = a_{i.k}^{max} - a_{i.k}^{min}$.

For instance, the queries from Example 5.9 have the following widths of intervals:

$width(a_1.1) = 20 - 10 = 10; width(a_1.2) = 50 - 40 = 10;$
$width(a_2.1) = 40 + 90 = 130; width(a_2.2) = 90 - 50 = 40;$

A set of values $A_i^{NA} = \{a_{i.1}^{NA}, \dots, a_{i.k}^{NA}\}$ valid with regard to a filtering condition $P_i[a]$ over a nominal attribute a is a set of values of a where each value satisfies the conditions put on a by the filtering condition $P_i[a]$. As for intervals, we introduce an extraction procedure:

Definition 5.12 The set of valid values of a filtering condition $P_i[a]$ associated with a nominal Attribute a $A_i^{NA} = \{a_{i.1}^{NA}, \dots, a_{i.k}^{NA}\}$ is as follows. If $P_i[a]$ is:

(1) Atomic clause: A_i^{NA} is a singleton set containing exactly the clause;
(2) Conjunction clause $(C_1 \wedge C_2) : A_i^{NA}$ is the intersection of valid values in C_1 and C_2;
(3) Disjunction clause $(C_1 \vee C_2) : A_i^{NA}$ is the union of valid values in C_1 and C_2;
(4) Negative clause (NOT C): A_i^{NA} is all the values from domain not presented in C.

5.1.4 Corner cases of SQL query similarity

To come up with a query similarity function, we ask:

(1) How to define it in the simplest case.
(2) How to quantify the similarity of two queries in the following cases:
  (a) There are several occurrences of an attribute in the filtering conditions in both queries.
  (b) There are different attributes in the filtering conditions, while the queries have at least one common attribute.
  (c) At least one query contains joins.

In what follows, we will answer these questions.

A distance function must meet the conditions from Section 5.1.1. We will prove that our distance/similarity function has these characteristics. The next section introduces some underlying notions of SQL query similarity.

5.1.5 The simplest case: two approaches of AAB similarities.

We first study the simplest case, when two queries have one occurrence of the same attribute in the filtering condition and nothing else. It seems plausible that similar queries are those whose access areas overlap. However, this might be too strict in certain cases.

Example 5.13 Think of a query log containing the queries:

q_1: **SELECT** * **FROM** Cities C **WHERE** C.latitude >= 45 **AND** C.latitude < 90

q_2: **SELECT** * **FROM** Cities C **WHERE** C.latitude >= 30 **AND** C.latitude < 45

q_3: **SELECT** * **FROM** Cities C **WHERE** C.latitude >= -75 **AND** C.latitude < -30

q_4: **SELECT** * **FROM** Cities C **WHERE** C.name = 'New York'

q_5: **SELECT** * **FROM** Cities C **WHERE** C.name = 'Paris'

Attribute latitude of table *Cities* has a continuous type, *Cities.latitude* $\in [-90; 90]$. This attribute is ordinal (OA), while *Cities.name* is nominal (NA). Now look at the first three queries in the log. Figure 5.13 plots the access areas of Queries q_1, q_2 and q_3. They are as follows:

$$q_1 : \sigma_{latitude \geq 45 \wedge latitude \leq 90}(Cities);$$
$$q_2 : \sigma_{latitude \geq 30 \wedge latitude \leq 45}(Cities);$$
$$q_3 : \sigma_{latitude \geq -75 \wedge latitude \leq -30}(Cities).$$

No two queries overlap. But q_1 and q_2 appear to be closer to each other: Their access areas even are adjacent.

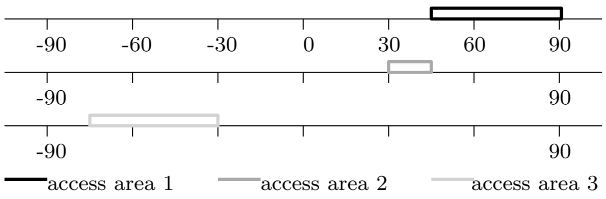

Figure 5.1: Access areas of attribute *Cities.latitude* for queries q_1, q_2 and q_3 from Example 5.13.

So we need to take in closeness as a criterion as well. All already existing measures which rely on the data, like *WB* or *AAB* from [Ngu+15], currently do not feature this. In other words, the phenomenon that closeness is neglected is not specific to access-area-based approaches.

5.1.5.1 Closeness Similarity for Ordinal Attributes

We want to quantify the closeness of the access areas of two queries. Lack of overlap of access areas does not mean 'zero similarity'. Put differently, SQL queries which request data in neighbouring parts of the data space should have the chance to end up in the same cluster.

Definition 5.14 The similarity of two queries with the same filtering ordinal attribute (OA) $S_{cl}(q_1, q_2).a$ is the proximity (*closeness, cl*) of their access areas:

$$S_{cl}(a_{1.i}^{OA}, a_{2.j}^{OA}) = \frac{1}{2} \cdot \frac{(a_{1.i}^{max} - a_{1.i}^{min}) + (a_{2.j}^{max} - a_{2.j}^{min})}{max(a_{1.i}^{max}, a_{2.j}^{max}) - min(a_{1.i}^{min}, a_{2.j}^{min})} \tag{5.2}$$

$a_{i.1}^{min}/a_{i.1}^{max}$ are the minimum/maximum of Interval $a_{i.1}^{OA}$.

Since we are considering the simplest case, there is only one interval of one attribute for each query. Because of this, the similarity of attribute conditions is the similarity of the first occurrence of Attribute a in both queries: $S_{cl}(q_1, q_2).a = S_{cl}(a_{1.1}^{OA}, a_{2.1}^{OA})$. The coefficient 0.5 normalizes the measure. The formula is the share of the space accessed over the width of the space between the queries. $S_{cl} > 0.5$ indicates overlap of access areas.

Lemma 5.15 $D_{cl}(q_1, q_2).a$ is semi-metric.

Proof. Moving from similarity to distance,

$D_{cl}(q_1, q_2).a = (a_{1.i}^{OA}, a_{2.j}^{OA}) = 1 - \frac{1}{2} \cdot \frac{(a_{1.i}^{max} - a_{1.i}^{min}) + (a_{2.j}^{max} - a_{2.j}^{min})}{max(a_{1.i}^{max}, a_{2.j}^{max}) - min(a_{1.i}^{min}, a_{2.j}^{min})}$

Symmetry: $D_{cl}(q_1,q_2).a = D_{cl}(q_2,q_1).a$

$D_{cl}(q_2,q_1).a = (a_{1.i}^{OA}, a_{2.j}^{OA}) = 1 - \frac{1}{2} \cdot \frac{(a_{1.i}^{max} - a_{1.i}^{min}) + (a_{2.j}^{max} - a_{2.j}^{min})}{max(a_{1.i}^{max}, a_{2.j}^{max}) - min(a_{1.i}^{min}, a_{2.j}^{min})}$

This is identical to $D_{cl}(q_1,q_2).a$. Thus, $D_{cl}(q_1,q_2).a = D_{cl}(q_2,q_1).a$.

Positivity: $D_{cl}(q_1,q_2).a \geq 0$

$D_{cl}(q_1,q_2).a \geq 0$ means $S_{cl}(q_1,q_2).a \leq 1$. The maximum similarity is achieved with query identity: $a_{1.1}^{min} = a_{2.1}^{min} \wedge a_{1.1}^{max} = a_{2.1}^{max}$. In this case,

$S_{cl}(q_1,q_2).a = \frac{1}{2} \cdot \frac{2 \cdot (a_{1.1}^{max} - a_{1.1}^{min})}{a_{1.1}^{max} - a_{1.1}^{min}} = 1.$

Hence $S_{cl}(q_1,q_2).a \leq 1$ is proven as well as $D_{cl}(q_1,q_2).a \geq 0$.

Reflexivity: $D_{cl}(q_1,q_2).a = 0$ iff $q_1 = q_2$

Since $q_1 = q_2$, $a_{1.1}^{min} = a_{2.1}^{min} \wedge a_{1.1}^{max} = a_{2.1}^{max}$. Hence,

$S_{cl}(q_1,q_2).a = \frac{1}{2} \cdot \frac{2 \cdot (a_{1.1}^{max} - a_{1.1}^{min})}{a_{2.1}^{max} - a_{2.1}^{min}} = \frac{1}{2} \cdot \frac{2 \cdot (a_{1.1}^{max} - a_{1.1}^{min})}{a_{1.1}^{max} - a_{1.1}^{min}} = 1$

Therefore, $D_{cl}(q_1,q_2).a = 1 - S_{cl}(q_1,q_2).a = 0.$ □

5.1.5.2 Overlap Similarity for Nominal Attributes

The closeness measure $S_{cl}(q_1,q_2).a$ in Equation 5.2 does not work with nominal attributes (*NA*). See Queries q_4 and q_5 from Example 5.13. The values of Attribute '*name*' of Table '*Cities*' do not have a natural order. To illustrate, 'Paris' is not close to 'Prague' just because they both start with 'P' *. Having said this, for nominal attributes, we propose to take the overlap (ovl) as the similarity. We use the Jaccard coefficient to this end:

$$S_{ovl}(q_1,q_2).a = \frac{|A_1^{NA} \cap A_2^{NA}|}{|A_1^{NA} \cup A_2^{NA}|} \tag{5.3}$$

In our case (Example 5.13),

$S_{ovl}(q_4,q_5).Cities.name = \frac{|\{'New\ York'\} \cap \{'Paris'\}|}{|\{'New\ York'\} \cup \{'Paris'\}|} = 0.$

Lemma 5.16 $D_{ovl}(q_1,q_2).a$, where a is a nominal attribute, is semi-metric.

Proof. The corresponding distance function is $D_{ovl}(q_1,q_2).a = 1 - S_{ovl}(q_1,q_2).a$. Since the Jaccard distance is symmetric, positive and reflexive, our measure has these characteristics as well. □

*In principle, one can use domain-specific ontologies and respective distance measures. To continue the example, Paris and Prague might be similar because they both are capitals of European countries with a rich history. However, taking such additional information into account is beyond the scope of this paper.

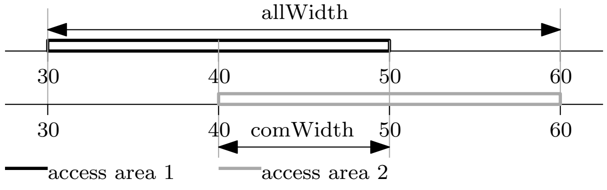

Figure 5.2: Access areas of attribute *Cities.latitude* from Example 3.8.

5.1.5.3 Overlap Similarity for Ordinal Attributes (OA)

With the definitions so far, we would rely on different paradigms, i.e., closeness and overlap, when calculating the similarity for ordinal and nominal attributes. When different types of attributes are treated differently, it is unclear how this will affect analysis results, e.g., clustering. Therefore, to have an alternative which we can use as a reference point later, we now propose a purely overlap-based similarity measure for ordinal attributes:

$$S_{ovl}(q_1,q_2).a = S_{ovl}(a_{1.1}^{OA},a_{2.1}^{OA}) = \frac{comWidth(a_{1.1}^{OA},a_{2.1}^{OA})}{allWidth(a_{1.1}^{OA},a_{2.1}^{OA})} \tag{5.4}$$

In each query, one interval $a_{1.i}^{OA}$ represents Attribute a. $comWidth(a_{1.1}^{OA},a_{2.1}^{OA})$ in Formula 5.4 is the overlap of Queries q_1 and q_2 for Attribute a in absolute terms:

$$comWidth(a_{1.1}^{OA},a_{2.1}^{OA}) = max(0, min(a_{1.1}^{max},a_{2.1}^{max}) - max(a_{1.1}^{min},a_{2.1}^{min})) \tag{5.5}$$

$allWidth(a_{1.1}^{OA},a_{2.1}^{OA})$ is the difference between the highest maximal bound and the lowest minimal bound:

$$allWidth(a_{1.1}^{OA},a_{2.1}^{OA}) = max(a_{1.1}^{max},a_{2.1}^{max}) - min(a_{1/1}^{min},a_{2.1}^{min}) \tag{5.6}$$

For instance, the similarity for Attribute Cities.latitude from Example 3.8 (from the literature review) is:

$$S_{ovl}(q_1,q_2).latitude = \frac{min(50,60)-max(30,40)}{max(50,60)-min(30,40)} = \frac{50-40}{60-30} = \frac{1}{3}$$

Figure 5.2 graphs the corresponding access areas.

Lemma 5.17 $D_{ovl}(q_1,q_2).a$, where a is an ordinal attribute, is semi-metric.

Proof. Distance $D_{ovl}(q_1,q_2).a = 1 - S_{ovl}(q_1,q_2).a$ is symmetric since we operate with interval lengths. It is also positive since the overlap is never bigger than the overall width. When two queries are identical, $comWidth(a_{1.1},a_{2.1}) = allWidth(a_{1.1},a_{2.1})$, and hence $D_{ovl}(q_1,q_2).a = 0$. Thus, $D_{ovl}(q_1,q_2).a$ is semi-metric. □

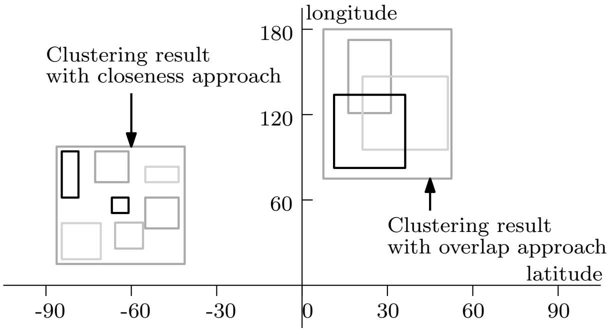

Figure 5.3: Clustering result for closeness or overlap approach

5.1.5.4 Summary

We have identified two paradigms of AAB query similarity: *closeness (AABcl)* and *overlap (AABovl)*. We use these acronyms from now on. We also have proposed query-similarity measures for ordinal attributes (Formula 5.2 for AABcl and 5.4 for AABovl) and nominal ones (Formula 5.3). Which method to apply (*closeness* or *overlap*) when it comes to ordinal attributes depends on the objective. Our hypotheses, which our experimental evaluation will address, are as follows: If an analyst is interested in exact access areas many users have looked for, he might want to use the "strict" overlap formula 5.4. In contrast, if he is more interested in the bigger picture, i.e., approximate, rather big areas users have looked at, the less strict closeness formula 5.2 might be better. Figure 5.3 shows the clustering results with the two approaches. Our experiments in Section 5.2 will provide more details.

So far, we have discussed similarity measures for the simplest case, two queries having the same attribute in the filtering conditions, and this attribute occurs only once. Now we turn to more complex cases.

5.1.6 Multiple Occurrences of an Attribute in Filtering Conditions

The first complication, when calculating query similarity, described at the beginning of Section 5.1, occurs when one uses the same attribute several times in the same query. This may happen when a query consists of OR predicates (for ordinal and nominal attributes) or IN predicates (for nominal attributes).

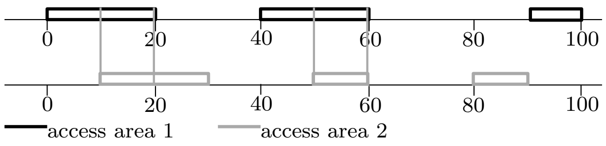

Figure 5.4: Access areas of queries with multiple occurrence of an attribute from Example 5.18

5.1.6.1 Overlap Similarity

Example 5.18 Consider the following queries:

```
q1: SELECT * FROM Cities WHERE (population BETWEEN 0 and 20)
      OR (population BETWEEN 40 and 60) OR (population BETWEEN 90 and 100)

q2: SELECT * FROM Cities WHERE (population BETWEEN 10 and 30)
      OR (population BETWEEN 50 and 60) OR (population BETWEEN 80 and 90)
```

The access areas of the queries look like in Figure 5.4. Both Queries q_1 and q_2 have not one, but several occurrences of Attribute Cities.population. Some of the areas do intersect: for instance, `(population BETWEEN 0 and 20)` of q_1 and `(population BETWEEN 10 and 30)` of q_2. Some however, like `(population BETWEEN 40 and 60)` of q_1 and `(population BETWEEN 80 and 90)` of q_2, do not.

With an overlap approach for ordinal attributes, we define the similarity measure as the ratio of the width of the overall overlap of the intervals to the width of the union of the intervals. Formally,

$$S_{ovl}(q_1,q_2).a = \frac{overallComWidth(a_1,a_2)}{overallAllWidth(a_1,a_2)} \tag{5.7}$$

The terms in the numerator and in the denominator are as follows:

$$overallComWidth(a_1,a_2) = \sum_{i=1,j=1}^{i=l_1,j=l_2} comWidth(a_{1.i}^{OA}, a_{2.j}^{OA}) \tag{5.8}$$

$$\begin{aligned} overallAllWidth(a_1,a_2) = \sum_{i=1}^{i=l_1} width(a_{1.i}^{OA}) + \sum_{j=1}^{j=l_2} width(a_{2.j}^{OA}) \\ - overallComWidth(a_1,a_2) \end{aligned} \tag{5.9}$$

where l_1 and l_2 are the numbers of intervals over Attribute a occurring in Queries q_1 and q_2 respectively. The width of an interval is calculated as in Definition 5.11.

Example 5.19 Think of the queries from Example 5.18. Here,

> $population_{1.1} = [0;20]$, $population_{1.2} = [40;60]$, $population_{1.3} = [90;100]$;
> $population_{2.1} = [10;30]$, $population_{2.2} = [50;60]$, $population_{2.3} = [80;90]$.
> $S_{ovl}(q_1, q_2).population = 20/(50+40-20) = 2/7$.

For nominal attributes, the formula remains exactly the same as in Section 5.1.5.2 (Formula 5.3). This is because it already covers several occurrences of an attribute.

Lemma 5.20 $D_{ovl}(q_1, q_2).a$, where a is an ordinal attribute which occurs several times in queries q_1 and q_2, is semi-metric.

Proof. As with the simple case, this distance is symmetric because it works with interval lengths. It is also positive because $overallComWidth(a_1, a_2)$ is always less than the sum of the widths of the intervals. For identical queries, $overallAllWidth(a_1, a_2) = overallComWidth(a_1, a_2)$– the distance is reflexive. Thus, the overlap distance $D_{ovl}(q_1, q_2).a$ is a semi-metric. □

5.1.6.2 Closeness Similarity

With the closeness similarity that we have considered so far, queries without overlap can be similar. Thus, Formula 5.7 is not applicable in this case. Hence, we propose to calculate overall closeness similarity for ordinal attribute (OA) a is as follows:

$$S_{cl}(q_1, q_2).a = \max_{i=1,\ldots,l_1; j=1,\ldots,l_2} S_{cl}(a_{1.i}^{OA}, a_{2.j}^{OA}) \tag{5.10}$$

The formula takes the maximum of pairwise similarities. So the closest intervals of two queries determine the similarity. Besides, 'max' has some desirable properties:

(1) It returns a normalized value in the [0;1] range of values. That is different from aggregation with, say, *sum*.
(2) It does not underestimate the similarity of two queries with the closeness paradigm. *min* or $\prod$ in turn do.

Lemma 5.21 $D(q_1, q_2).a$ is semi-metric.

Proof. Formula 5.10 uses Formula 5.2 (simple case). We have already proven that this is a semi-metric. With aggregation with *max*, the distances remain positive, symmetric and reflexive, i.e., this similarity measure is a semi-metric as well. □

5.1.7 Several Distinct Attributes in Filtering Conditions

So far, we have discussed the cases when both queries filter with the same single attribute: $Ints(q_1) = Ints(q_2); |Ints(q_1)| = |Ints(q_2)| = 1$. The following example illustrates the case of different attributes in the filtering conditions of two queries.

Example 5.22 A query log contains the following queries:

q_1:
```
SELECT * FROM Cities C WHERE C.latitude BETWEEN 52 and 80
    AND C.longitude BETWEEN 30 and 45 AND C.population BETWEEN 30 and 500
```

q_2:
```
SELECT * FROM Cities C WHERE C.latitude BETWEEN 40 and 52
    AND C.longitude BETWEEN 30 and 45 AND C.country ='France'
```

Queries q_1 and q_2 have two common interests:
$comInts(q_1, q_2) = \{C.latitude, C.longitude\}$.

They also have exclusive interests:
$exclInts(q_1, q_2) = \{C.population, C.country\}$.

We propose to calculate the similarity measure $S_{attrFull}(q_1, q_2)$ where both $comInts(q_1, q_2)$ and $exclInts(q_1, q_2)$ are considered with similarities $S_{attrCom}(q_1, q_2)$ and $S_{attrExcl}(q_1, q_2)$. Section 5.1.9 will feature the concrete formula.

5.1.7.1 Similarity for Common Interests

Till this moment we have defined the attribute-wise similarity for queries $S(q_1, q_2).a$. If queries however have more than one common interest, as in Example 5.13, we need a definition which takes the attribute-wise similarities for all common attributes $S(q_1, q_2).a, a \in comInts(q_1, q_2)$, as input. Since a query may contain ordinal and nominal attributes, we cannot separate closeness and overlap approaches. Instead, a general, unifying approach to arrive at meaningful overall similarities is necessary. Coming back to Example 5.22,

$$S_{cl}(q_1, q_2).latitude = 0.5; S_{ovl}(q_1, q_2).latitude = 0$$
$$S_{cl}(q_1, q_2).longitude = 1; S_{ovl}(q_1, q_2).longitude = 1$$

The overlap approach assumes that, if there is no overlap, then there is no similarity. Any non-overlapping condition should lead to zero similarity. For our example, $S_{ovl}(q_1, q_2) = 0$ since $S_{ovl}(q_1, q_2).latitude = 0$. In general,

$$S_{attrCom}(q_1, q_2) = \min_{i=1,\dots,|comInts(q_1,q_2)|} S(q_1, q_2).a^i \tag{5.11}$$

where a^i is an attribute contained in $comInts(q_1, q_2)$. *min* does not overestimate the similarity. Hence, one might expect relatively small clusters with clear user interests.

Since we do not see any alternative how the overlap approach could be generalized, we use Formula 5.11 for the closeness approach as well.

Lemma 5.23 $D_{attrCom}(q_1, q_2)$ is semi-metric.

Proof. With aggregation by means of min, the corresponding distance remains a semi-metric. □

5.1.7.2 Similarity for Exclusive Interests

If two queries have at least one shared interest, but also have exclusive ones, the similarity measure should reflect this. For each attribute in $exclInts(q_1, q_2)$, we calculate an overlap similarity value. We assume that an empty filtering condition in q_1 or q_2 means that one is interested in the entire domain of that attribute.

$$S_{attrExcl}(q_1, q_2) = \min_{i=1,\ldots,|exclInts(q_1,q_2)|)} S_{ovl}(q_1, q_2).a^i \tag{5.12}$$

Here, a^i is an attribute contained in $exclInts(q_1, q_2)$. In fact, $S_{attrExcl}(q_1, q_2)$ just calculates overlap similarity among attributes that do not occur in both queries.

Example 5.24 Let us now calculate $S_{attrExcl}(q_1, q_2)$ for queries from Example 15.22. Suppose that $C.country$ has 250 distinct values, and that $C.population \in [0; 20000]$.

$$exclInts = C.country, C.population;$$

$$S_{ovl}(q_1, q_2).country = \frac{|\{'France'\}|}{|\{'Afghanistan', \ldots, 'Zimbabwe'\}|} = 0.004;$$

$$S_{ovl}(q_1, q_2).population = (500 - 30)/20000 = 47/2000 = 0.0235;$$

$$S_{attrExcl}(q_1, q_2) = min(0.004, 0.0235) = 0.004$$

There are two reasons for using overlap-based similarity for $S_{attrExcl}$:

(1) A non-shared filtering attribute can be nominal. We do not see any reason why non-shared ordinal and nominal attributes should be treated differently.
(2) We believe that dissimilar interests stand for different user intentions. The similarity values should be low. The closeness approach for ordinal attributes might yield clusters of queries without similar interests.

Lemma 5.25 $D_{attrExcl}(q_1, q_2)$ is semi-metric.

Proof. Similarly to $S_{attrCom}(q_1, q_2)$, with min as aggregation function, the corresponding distance remains to be semi-metric. □

5.1.7.3 Overall Attribute Similarity

Finally, the minimum of $S_{attrExcl}(q_1, q_2)$ and $S_{attrCom}(q_1, q_2)$ is the overall similarity of attributes:

$$S_{attrFull}(q_1, q_2) = min(S_{attrExcl}(q_1, q_2), S_{attrCom}(q_1, q_2)) \tag{5.13}$$

Lemma 5.26 $D_{attrFull}(q_1, q_2)$ is semi-metric.

Proof. The distance function is still semi-metric since we use min as the aggregation function. □

All predicates including join predicates are processed when we compute attribute similarities (Formula 5.13). While we mostly use one- or two-dimensional examples, the principle is independent from this number.

5.1.8 Similarity in the Presence of Joins

The last remaining difficulty regarding our AAB similarity function is what needs to be done in the presence of joins.

Example 5.27 Consider the following query log:

q_1:
```
SELECT * FROM Cities C INNER JOIN Objects O ON C.objId = O.objId
    WHERE O.latitude BETWEEN 52 and 80 AND O.longitude BETWEEN 30 and 45
```

q_2:
```
SELECT * FROM PowerStations PS INNER JOIN Objects O ON PS.objId = O.objId
    WHERE O.latitude BETWEEN 52 and 80 AND O.longitude BETWEEN 30 and 45
```

q_3:
```
SELECT O.id FROM Objects O
    WHERE O.latitude BETWEEN 52 and 80 AND O.longitude BETWEEN 30 and 45
```

q_4:
```
SELECT O.id, T.typeName FROM Objects O INNER JOIN Types T ON O.type = T.id
    WHERE O.latitude BETWEEN 52 and 80 AND O.longitude BETWEEN 30 and 45
```

Queries q_1 and q_2 look for different objects, i.e., entities from different relations, but in the same part of the data space. One must take this distinction into account. But our metric so far only relies on the filtering conditions.

An intuitive solution is to multiply the overall attribute-similarity values $S_{attrFull}(q_1, q_2)$ with a value quantifying the overlap of the sets of tables accessed, e.g., the Jaccard coefficient:

$$S_{final}(q_1, q_2) = \frac{|q_1.FROM \cup q_2.FROM|}{|q_1.FROM \cap q_2.FROM|} \tag{5.14}$$

where $q_i.FROM$ is the set of tables accessed by Query q_i. This approach, while being simple, has a problem. Consider Queries q_3 and q_4. They search all objects within identical coordinate boundaries, i.e., intervals. q_4 has as additional output the type of an object which comes from the join with Table Types. According to Formula 5.15, the JOIN in Query q_4 decreases the similarity of q_3 and q_4 twice, from 1 to 0.5. Adding more joins to q_4, just to output more information, reduces similarity even more, compared to the query without joins. Hence, we argue that an adequate reduction coefficient should consider the size of tables accessed, in rows:

$$S_{final}(q_1,q_2) = reductCoeff_{table} \cdot S_{attrFull}(q_1,q_2) \tag{5.15}$$

$$reductCoeff_{table} = \frac{\sum_{i=1}^{i=|T_{com}|}}{\sum_{j=1}^{j=|T_{all}|}} \tag{5.16}$$

Here, T_{com} is the set of common tables of Queries q_1 and q_2:

$T_{com} = q_1.FROM \cap q_2.FROM.$

Accordingly, T_{all} is the set of all tables accessed in Queries q_1 and q_2:

$T_{all} = q_1.FROM \cup q_2.FROM.$

Example 5.28 Let us suppose that $Types.size = 20$, $Objects.size = 980$.
The reduction coefficient $reductCoeff_{table}$ for q_3 and q_4 is:

$T_{com} = q_3.FROM \cap q_4.FROM = \{'Objects'\};$
$T_{all} = q_3.FROM \cup q_4.FROM = \{'Objects', 'Types'\};$

$reductCoeff_{table} = \frac{Objects.size}{Cities.size+Objects.size} = \frac{980}{20+980} = \frac{49}{50}.$

$reductCoeff_{table}$ has a specific value for each query pair. One should apply it once after having calculated the overall attribute similarity $S_{attrFull}(q_1,q_2)$.

Lemma 5.29 $D_{final}(q_1,q_2)$ is semi-metric.

Proof. After multiplying by $reductCoeff_{table}$, the overall distance remains semi-metric. □

Recall that the semi-metric characteristic is helpful: It allows us to use our similarity/distance function in many clustering algorithms without any further validation.

5.1.9 The Overall AAB Similarity Function

Summarizing what we have said so far, the overall AAB similarity function is as follows:

$$S(q_1, q_2) = reductCoeff_{table} \cdot S_{attrFull}(q_1, q_2) \tag{5.17}$$

To calculate the similarity $S(q_1, q_2).a$ for an attribute which exists in both queries, one can use Formulas 5.7 or 5.10, depending on the approach, i.e., AABovl or AABcl. Consequently, Formulas 5.7 and 5.10 refer to the simple case of Formulas 5.2, 5.3 and 5.4.

5.1.10 Discussion

In a nutshell, the AAB query representation captures parts of the data space where the user has an interest in. The WB query representation has the same objective, by identifying the relevant data explicitly. Hence, we expect to get similar results from clustering. However, as we have already pointed out, WB lacks scalability. AAB in turn does not have this limitation since it operates with access areas, not the data itself.

5.2 Experimental evaluation

This section evaluates various algorithms, query representations and similarity functions for query-log clustering. Our objectives are:

(1) Investigate the precision of the various QRSs (FB, WB, AAB) experimentally, on data where a ground truth is available. A QRS is precise if it leads to a clustering with a big overlap with the ground truth;
(2) Generate clustering results with real-world data, including the SkyServer query log in our case, inspect it and try to arrive at general insights;
(3) Study the influence of sampling on the clustering result.

5.2.1 Experiment Settings

The quality of any clustering is hard to evaluate without a ground truth. One can ask domain experts to provide an interpretation of the results. However, in our current context, a domain expert may not be able to say whether a result is good or even perfect – there often does not exist any expectation how an ideal result should look like. To cope with this problem to some extent, we propose two experiments. The first one is clustering queries which hundreds of individuals have formulated to solve a specific task. In our case, these individuals are university students participating in a database course. They have solved this task as part of an exercise where the information need was given in natural language. We have anonymized the data before our analysis so that it did not contain any personal information. This experiment 'only' serves to study the precision of the various query representations, by comparing clustering results to a ground truth. This is because the data set available is relatively small, and it has turned out that some query representations are more sensitive to it than another. Ideally, all queries that the students have formulated to solve a particular task should end up in one cluster.

The second experiment is a case study with a real-world SQL log. We use the SkyServer query log, a large log of queries on scientific data available to the public. To our knowledge it is the only query log publicly available. More specifically, we use the SkyServer log for 2016[†]. It consists of 12.9 million queries from about 4,000 users.

For each data we first calculate pairwise distances, i.e., build a proximity matrix. There are well-established types of clustering methods which can work with a proximity matrix: hierarchical clustering, partitioning-based methods and density-based approaches. To make our study comprehensive, we select well-known instances from the different categories of clustering methods. We choose the hierarchical algorithm CLINK [Def77], k-medoids [PJ09] from the class of partitioning-based methods and DBSCAN [Est+96] as a density-based approach.

[†]We have decided to use the newer (compared to data from Chapter 4) fraction of the log to be more up-to-date with our results

Table 5.1: Description of the GtDbCourseLog data

Property	**Overall**	**WB**	**FB**	**AAB**
Size of original query log	1062	-	-	-
Preprocessed queries	-	610	659	647
Among preprocessed				
Task 1		259	259	252
Task 2		66	119	111
Task 3		228	221	227
Task 4		57	60	57
Non-preprocessed queries	-	452	403	415
Among non-preprocessed				
No solution provided	129	129	129	129
JSqlParser error	60	60	60	60
Syntax errors	214	214	214	214
Empty result set	-	64	-	-
Empty WHERE clause	-	-	-	12

5.2.2 The Data Sets

In this section, we describe the data sets for our experiments – the small one from the student exercise and the big one from the SkyServer project.

5.2.2.1 Data Set from the Student Exercise

We have collected 1062 SQL requests formulated by 274 participants of the database course at our institution in the summer semester 2016. To facilitate repeatability, we make these queries and the test database publicly available . We have asked the participants to produce solutions to four information needs. Thus, our ground truth is that all solutions to one information need form a cluster. Hence, we expect 4 clusters. Of course, not all answers have been complete and syntactically correct. Table 5.1 of the Appendix is a summary of GtDbCourseLog, the log with these queries from the database course.

We have considered using other labeled query logs: [Cha+15] and [Kul+18]. [Cha+15] however is not publicly available. [Kul+18] is the log of a database exam. First, it has only two tasks, i.e., one may expect two clusters. It is smaller than in our log. Second, the log is smaller in terms of the number of queries as well (178 against 1062). Moreover, [Kul+18] does not include the test database, which the WB query representation requires. So we have decided to use only one query log with a ground truth, *GtDbCourseLog*.

5.2.2.2 SkyServer Query Log

The original SkyServer query log for 2016 consists of 12.9 million queries. The clustering procedure considers only queries which have both an AAB and an FB

Table 5.2: Description of the SkyServer Log data

Property	**Value**	**Share**
Size of original query log	12,917,940	
Preprocessed queries (FB, AAB), SkSLog	10,289,990	79.7 %
Non-preprocessed queries (FB, AAB)	2,627,950	20.3 %
Among non-processed		
Arithmetic operation in WHERE clause	1,158,375	9.0 %
JSQL Parser limitations	784,798	6.1 %
Queries to meta-tables	364,505	2.8 %
Queries without WHERE clause	344,552	2.7 %
Errors in SkyServer logging	17,956	0.1 %
Queries to non-existing tables	2,444	0.02 %
Cleaned queries, FullLog	1,368,232	10.6 %
Queries excluded	8,921,758	69.1 %
Among excluded		
Requests made by robots performing SWS	8,001,943	62 %
Requests which refer to SkyServer web pages with default values	919,815	7.1 %

query representation. Having the same input data for a comparison of different query-representation schemes is a prerequisite for meaningful results. As mentioned in Section 3.3.4, the current version of AAB does not process queries with arithmetic operations in the WHERE clause, and we also exclude SQL requests without a predicate in the filtering condition. Table 5.2 summarizes the queries included in the comparison and contains explanations for queries which we have not processed.

With query clustering, one wants to obtain meaningful results, i.e., finding user interests in our case. So we also exclude queries with the following characteristics from further consideration:

(1) *Queries issued by robots performing a sliding window search (SWS).* To identify this behaviour, we have run a procedure described in Chapter 4. To identify SWS, we use the FB and AAB query representations. The procedure consists of two steps:

 (a) From the FB query representations, we get queries which might be SWS: more than 100 the same FB representations from one user.

 (b) We filter out user sessions which do not perform range queries. To do so, we check the corresponding intervals in the AAB query representations. A user session $US = q_1, \dots, q_n$ is an SWS iff $\forall q_i \in US$:

  i. $q_i.intl_{min} \neq q_i.intl_{max}$

  ii. $q_i.intl_{min} \neq q_{i+1}.intl_{max}$

  iii. $q_i.intl_{max} \neq q_{i+1}.intl_{max}$

 Here, $q_i.intl_{min}$ and $q_i.intl_{max}$ are the lower and upper bounds of an interval in Query q_i.

As defined earlier, an SWS is a sequence of queries of identical structure, performing a range search (see Section 4.3.5). Here, identical structure means that only parameter values are different, and the ranges are contiguous.

Table 5.3: Description of the WB sampled data (SampledLog)

Property	**Value**
Size of WB sampled log	137,101 (1.06%)
Number of WB processed queries	72,817 (0.56%)
Among non-processed	64,284
Queries which return empty result	18,089
Queries which were timed out or returned an error	46,195
Number of witnesses	288,379,030
Average number of witnesses	3,960
Number of distinct witnesses	109,830,577
Overall time elapsed for requerying	19.8 days
Overall processing time	22.3 days
The size of WB sampled log data	0.08 GB
The size of preprocessed data	8.19GB

How SWS queries – if included – affect clustering, de-pends on the similarity function used, AABovl or AABcl. In the case of AABovl, SWS queries do not form a cluster, because SWS imply disjoint filtering conditions (no overlap). Hence, for AABovl it counts as noise – more queries are processed, which increase the runtime. With AABcl, SWS queries could form a cluster, because "neighbour" queries will get non-zero similarities. In our opinion, this is not a result one needs to get since SWS represent the information need only of one user, not the common interest of many people.
The procedure of excluding SWS requires a threshold value as a parameter which specifies the strictness when looking for SWS. Here, we fix the value to 100, i.e., 100 contiguous range queries from one user in a row are an SWS. This kind of queries occupies 62% of the SkyServer query log. See Table 5.2. We for our part leave aside such queries since they represent interests of very few users or even only one, and it even is unclear what the true interest behind an SWS is.

(2) Queries issued by many users when they open the SkyServer web interface for the first time. To illustrate,
```
SELECT ...FROM fGetNearbyObjEq(258.25,64.05,3) n,
PhotoPrimary p WHERE n.objID=p.objID
```
has been issued 647907 times. It is available at the radial search web page of SkyServer, with exactly these default values. Thus, the full log after cleaning, named FullLog, consists of 1.37 million queries.

Obtaining WB query representations, even for a log of 1.3 million queries is difficult to impossible. As mentioned, one would need to evaluate all queries to this end. The overall runtime for all queries from the full cleaned log, the FullLog, is around 220 days according to SkyServer metadata, the sum of the numbers of rows in all results is about 7.7 billion. Thus, for the WB query representation, we have sampled FullLog, obtaining SampledLog. For this sampling, we have chosen one-tenth of the cleaned log. Table 5.3 is a description of the WB sampled dataset. From now on, we will use the names for the different query logs as in Table 5.4.

Table 5.4: Overview of the data actually used in the experiments

The log name	Description	Size
GtDbCourseLog	The log from the database course with a ground truth	1,062
SkSLog	The processed SkyServer log for 2016	10,289,990
FullLog	Cleaned SkSLog, without SWS and requests which refer to SkyServer web pages with default values	1,368,232
SampledLog	One tenth of FullLog (random sampling).	137,101

5.2.3 Evaluation Techniques

According to [HBV02], validity measures for clustering fall into two groups:

(1) *External* measures are used when ground truth is available. The idea behind these measures is to compare the clustering result with the ground truth. An example of such a measure is the *Jaccard index* [WW07]. The set of all pairs of objects from two clustering results C and C' is the disjoint union of the following sets:
   (a) $S_{11} = \{$pairs that are in the same cluster under C and $C'\}$
   (b) $S_{00} = \{$pairs that are in different clusters under C and $C'\}$
   (c) $S_{10} = \{$pairs that are in the same cluster under C but in different ones under $C'\}$
   (d) $S_{01} = \{$pairs that are in different clusters under C but in the same under $C'\}$

   The values n_{11}, n_{00}, n_{10} and n_{01} are the cardinalities of these sets. The Jaccard index now quantifies the similarity of two clustering results as follows:

$$J(C,C') = n_{11}/(n_{11}+n_{10}+n_{01}) \tag{5.18}$$

   The index takes values from 0 to 1. The bigger it is, the higher is the similarity.

(2) *Internal* measures do not require a ground truth. They rely on criteria derived from the data itself, e.g., intracluster and intercluster distances. In our internal evaluation, we use the *BetaCV* measure [ZMJM14], the ratio of the mean intercluster distance over the mean intracluster distance. A small value indicates higher clustering quality. To validate the consistency within clusters, we have used the Silhouette coefficient $s(i)$ [Rou87], which indicates how well each object i lies within its cluster. $s(i)$ takes values from -1 to 1. The bigger $s(i)$, the better i matches its cluster.

5.2.4 Implementation

To get the query representations, we first parse the queries. We use the JSQL Parser[‡] written in Java. We then stored query representations in the database. We have

[‡]http://jsqlparser.sourceforge.net/

implemented all similarity functions in SQL. To evaluate the results (i.e., to calculate the BetaCV coefficient etc.), we have again used Java.

5.2.5 Experiments with Supervision

Regarding the student exercise, since the students were asked to perform four tasks, we expect to get four clusters in the results. Hence, it may make sense to use the *k-medoids* clustering algorithm with $k = 4$ for each similarity function and corresponding query representation. Table 5.5 contains the results. As for similarity measures, we have used the Jaccard index and the BetaCV coefficient.

The Jaccard indexes for the WB and the FB query representation, which indicate the closeness of the clusters to the ground truth, do not show good results relative to the other approaches. We see three reasons for this:

(1) The database schema consists of only three tables. The tasks have been constructed to have more than one table in an answer SQL statement. There also are only a few attributes in each relation, namely 3, 7 and 3. Hence, the probability of having the same tables and filtering attributes for different tasks is high. That causes a problem with the FB approach. The value of 0.454 of BetaCV for the FB query representation indicates this.
(2) The database has been tiny as well – it has 28 rows from 3 tables. That leads to a high probability that queries from different tasks share the same tuples. That, in turn, affects WB clustering.
(3) The students have made mistakes when formulating queries. If all answers were precise, we would have obtained zero BetaCV for each query representation: While the text of two correct answers might differ, all query representations for one task would be identical. For instance, when looking at the WB representation, a correct query should return only certain tuples. The same holds for FB and AAB. That would have lead to zero intercluster distances. However, the actual average intercluster distances are above zero. See Table 5.5.

Since the AAB query representation does not rely very much either on metadata (database schema) or on actual data (witnesses), the respective clustering corresponds to the ground truth very well. We have obtained identical results for both AABovl and AABcl because of the high specificity of the tasks: Everybody has been asked the same, and mistakes by formulating wrong filtering conditions are unlikely. However, the AAB query representation (as well as FB and WB) cannot cope with the third problem (errors in the student answers). That is in line with our expectations.

To sum up, this experiment shows that all query representations lead to meaningful clustering in theory. However, there are certain obstacles which have turned out to be spoilers: These are the small database schema for FB and the tiny database for WB. On the other hand, the results are in line with the limitations we have already discussed when introducing the query representations.

Table 5.5: The results of the experiments with ground truth, dataset GtDb-CourseLog, clustering algorithm k-medoids, k = 4

	WB	FB	AABovl	AABcl
Jaccard index (compared with ground truth)	0.7518	0.4339	0.9451	0.9451
Experiment				
BetaCV	0.257	0.454	0.1402	0.1402
Average intercluster distance	0.194	0.317	0.1402	0.1402
Average intracluster distance	0.753	0.698	1	1
Average Silhoette coefficient	0.885	0.409	0.87	0.85
Ground truth				
BetaCV	0.171	0.194	0.231	0.231
Average intercluster distance	0.132	0.167	0.231	0.231
Average intracluster distance	0.773	0.857	1	1
Average Silhoette coefficient	0.74	0.754	0.689	0.689

5.2.6 Experiments with SkyServer

With SkyServer, we have generated the three query representations described before for the randomly sampled log, dubbed SampledLog. We first discuss the results generated from these. We also have experimented with FullLog, for the FB and AAB representations, and we describe it as well. Next, by comparing the results from the sampled log to the ones from the regular one, we evaluate how sampling affects the clustering results. Finally, we conduct a study with a domain expert to interpret our findings. All these experiments yield different insights regarding the usefulness of the query representations and the appropriateness of the various clustering algorithms when applied to a real-world SQL query log.

5.2.6.1 Clustering Results

Table 5.8 lists the parameter values for DBCSAN, k-medoids and CLINK. To set them, we rely on the expectation that the size of clusters should be in line with the size of input data. Thus we have set the value of parameter minPts of DBSCAN to 0.05% of the number of queries. The size of a dataset is the number of distinct objects which have at least one non-zero similarity value in the corresponding proximity matrix. That is because only these objects have a chance to be contained in a cluster. If the data is noisy, and there are many objects without similar ones, no group of similar objects is big enough to form a cluster. For the experiments with full data, FullLog, we set $minPts = 100$. We want to compare both the AABcl and the AABovl approaches with the same non-strict parameters, i.e. when $minPts = 100 < 0.05\%$ of the number of objects in both cases. We set $eps = 0.7$ for DBSCAN, since it captures sufficient overlap for the AABovl and the WB approach and allows to catch queries close to each other with AABcl. The number of clusters DBSCAN returns will be the value of our parameter k when running k-medoids. While the actual number of clusters is typically not available in most real settings, we use it here nevertheless. That is because we are interested in how good the results can be. For the hierarchical CLINK algorithm, the cut-off threshold is equal to *eps* of DBSCAN.

For the FB approach, however, we follow another strategy. Since FB yields templates of user behaviour, it does not make sense to mix several templates. Therefore we have chosen the parameters so that only very similar templates ($eps = 0.1$) go to the same cluster. Our clustering results consist of almost 1000 clusters: 508 for AABovlFull, 82 for AABovlSampled, 182 for AABclFull, 88 for AABclSampled and 125 for WB.

Table 5.7 lists the values of the BetaCV coefficient. They are relatively large. That is because we have considered only large clusters for their calculation; the size of a cluster must be no less than 0.05% of the size of a query log. Within such big clusters, a lot of queries have zero similarity with each other. That means that a Query q has an overlap with only a few queries $\{q'_1, \ldots, q'_n\}$. They are similar to other ones $\{q''_1, \ldots, q''_m\}$, though q is not similar to them. In other words, the clusters are not dense. Indeed, they cannot be since the corresponding proximity matrices are sparse. See Figure 5.5 with the similarity distributions. Table 5.6 reports on average Silhouette coefficients.

Nevertheless, for AABovl and WB, clustering yields areas of interests which are small compared to the whole data space. Tables 5.9 and 5.10 list aggregated query representations of the biggest clusters (in terms of number of queries) of DBSCAN, with SampledLog and FullLog. The representation is the minimum bounding rectangle (MBR) of all queries in the cluster. To present the results of WB clustering, one cannot utilize the corresponding query representations – they contain huge numbers of tuples. Instead, we use the more compact and intuitively understandable AAB representation as well. For AAB there also is an area-coverage value available. It is the ratio of the volume of the aggregated access area over the volume of the tables the queries from the cluster are applicable to:

$$areaCoverage = \frac{V_{access}}{V_{content}} \tag{5.19}$$

To obtain V_{access}, we take bounds of each attribute occuring in a filtering condition of the cluster. So $V_{content}$ is calculated taking the domains of each such attribute.

5.2.6.2 Discussion of Query Representations

We discuss the usefulness of the various query representations when clustering a real-world query log based on the experiments with SkyServer.

WB clustering We observe that WB clusters are precise. That is because all queries in the WB clusters ask for the same attributes, the spatial attributes *dec* and *ra*. That means that there have not been any "accidental" similarities, i.e., queries which refer to different attributes sharing witnesses by chance. With these identical attributes, queries whose filtering conditions overlap are similar.

AABovl clustering

(1) We find it remarkable that the aggregated access areas for AABovl and WB similarity are very much alike. Three of the four biggest (in a number of queries) clusters with these approaches point to the same parts of the sky. We conclude that the AAB query representation and AABovl similarity function also are valid and precise. With AAB being scalable, we for our part find that it may be preferable to WB.

(2) A difference we have observed in the AABovl and WB clustering results (with SampledLog) is that there are clusters in AABovl which do not exist in WB. The queries inside these clusters have empty results. Of course, WB cannot detect them. For example, the fifth biggest cluster of AABovl has the following aggregated access area:

$$\texttt{photoprimary.dec} \geq \texttt{-7.073} \wedge \texttt{photoprimary.dec} \leq \texttt{-7.026} \wedge$$
$$\texttt{photoprimary.ra} \geq \texttt{78.1498} \wedge \texttt{photoprimary.ra} \leq \texttt{78.195}$$

Indeed, there is no data object in this area. However, this has not prevented a significant number of AAB query representations from forming a cluster.

One might wonder why clusters with an overlap (like #2 and #3 of AABovl FullLog, see Figure 5.6) are not one single cluster. We had a closer look at this phenomenon and have observed that queries from one cluster #2 indeed are similar to SQL requests from the other cluster #3. However, these are points that are density-reachable, not core points in DBSCAN terminology. To conclude, density reachability is a characteristic that is not sufficient to end up in the same cluster in general.

AABcl clustering We have obtained big clusters which cover significant parts of the data space when clustering with the AABcl similarity function. That is plausible: As Figure 5.6 shows, the third biggest cluster of AABcl in the sampled log (the one with Rank 3) has a big rectangular part from to 41.269 to 84.973 in the *dec* column. This part is due to several queries with broad diapasons: These queries request data based on attributes *ra* and *dec* with broad ranges. Different users have issued these queries, so they are not SWS. They act like "*supermassive*" objects and have a "*gravity* *effect*" on queries with smaller ranges in the neighbourhood. In contrast to AABovl, where supermassive objects do not have sufficient overlap with small objects to fall into a cluster, AABcl is sensitive to queries with broad ranges. It is also sensitive to sliding window search (SWS). However, because we had filtered them out beforehand, we did not observe the influence of SWS on AABcl clustering. Summing up, whether AABcl is successful strongly depends on specifics the query log: It needs to be free from massive downloading, i.e., SWS, and there should not be any very broad range queries. Put differently, that also indicates that cleaning the query log before analysis might yield better, more meaningful results.

FB clustering Clustering, in line with the *FB* paradigm, reveals patterns of SkyServer database usage, i.e., which tables, views, UDFs and filtering attributes individuals tend to use. However, as mentioned before, this query representation does not reveal areas of the *data space* users are interested in. That also is why the column "Area coverage" is empty for *FB* clustering.

5.2.6.3 Discussion of clustering algorithms

Different clustering algorithms have performed differently on the SkyServer log as well. The data for the AAB and WB approaches contains noise – queries which do not have sufficiently many similar objects. Different algorithms have unlike sensitivity to this kind of noise. DBSCAN can work with this noisy data [19]. k-medoids suffers from it a lot since it partitions the data, and all objects end up in some cluster. CLINK is sensitive to noise as well, but ignoring small clusters can solve the problem here: If the data to be clustered contains a lot of outliers, many small or even singleton clusters occur. The algorithm does not merge them to bigger clusters since they are too distant from each other. So we have classified clusters with a size less than a specific value as noise. That is why the clustering result with CLINK does look structurally similar to the one with DBSCAN, containing an extra "cluster" for noise.

Summing up, we would give preference to a density-based clustering algorithm when it comes to query logs, for the following reasons:

(1) Data might be noisy.
(2) One cannot predict the number of clusters in advance, as required by k-means-based algorithms.

Consequently, we have clustered the big log file, FullLog, only with DBSCAN.

‖'pp' stands for 'photoprimary'
‖'po' stands for 'photoobj'
‖'poa' stands for 'photoobjall'
‡‡'pp' stands for 'photoprimary'
‡‡'poa' stands for 'photoobjall'
‡‡'as' stands for 'apogeestar'

Table 5.6: Values of Average Silhoette coefficient

Dataset	Algorithm	WB	AABovl	AABcl
SampledLog	DBSCA	**0.455**	0.43	0.098
	K-medoids	**0.008**	0.005	0.003
	CLINK	0.624	**0.721**	0.632

Table 5.7: Values of BetaCV coefficient

Dataset	**Algorithm**	**WB**	**AABovl**	**AABcl**
SampledLog	DBSCAN	0.933	0.925	0.993
	K-medoids	0.9995	0.9997	0.9998
	CLINK	0.998	0.997	0.999
FullLog	DBSCAN	-	0.913	0.981

Table 5.8: The parameter values for the clustering algorithms

Dataset	**Algorithm**	**Parameter**	**AABcl**	**AABovl**	**WB**	**FB**
SampledLog	DBSCAN	minPts	30	38	12	1
		epsilion	0.7	0.7	0.7	0.1
	K-medoids	K	90	71	124	592
	CLINK	cutting threshold	0.7	0.7	0.7	0.1
FullLog	DBSCAN	minPts	100	100	-	1
		epsilion	0.7	0.7	-	0.1

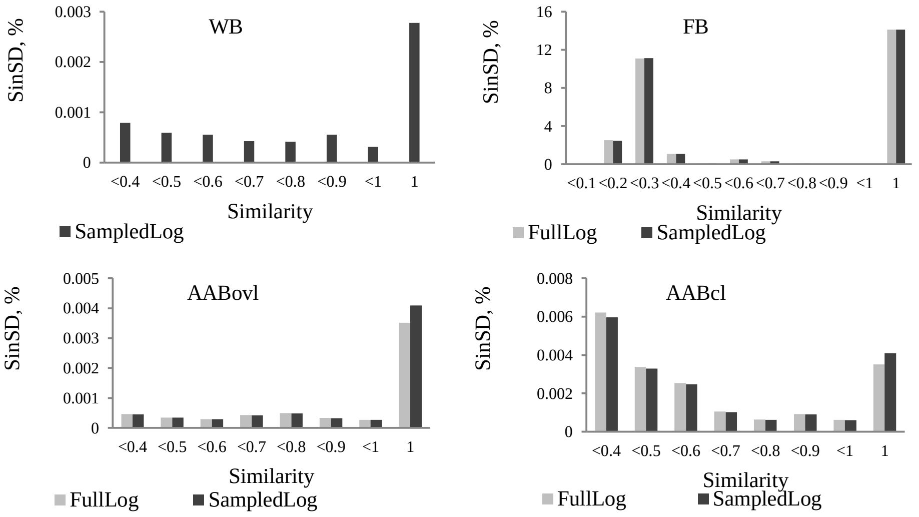

Figure 5.5: Similarity distributions, SinSD – Share in Similarity Distribution

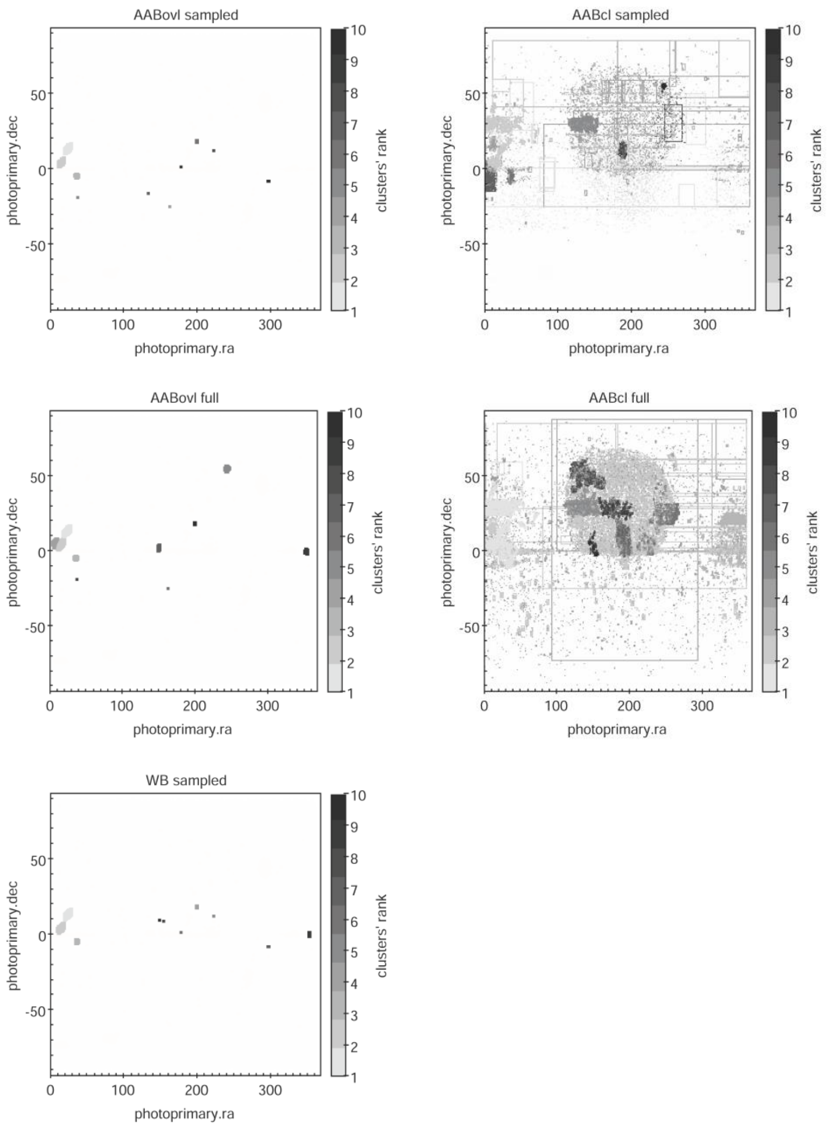

Figure 5.6: Clustering results, DBSCAN algorithm

Table 5.9: Top clusters of DBSCAN, dataset SampledLog

#	**Relative size, %**	**Area coverage, %**	**Aggregated query representation**
AABovl			
1	0.78	0.25	pp.dec[§] ≥ 1.2 ∧ pp.dec ≤ 7.3 ∧ pp.ra ≥ 10.3 ∧ pp.ra ≤ 18.8
2	0.08	0.02	pp.dec ≥ 54.8 ∧ pp.dec ≤ 56.8 ∧ pp.ra ≥ 241.4 ∧ pp.ra ≤ 245
3	0.07	0.002	pp.dec ≥ -9.1 ∧ pp.dec ≤ -9.05 ∧ pp.ra ≥ 120 ∧ pp.ra ≤ 120.05
4	0.06	$3.7 \cdot 10^{-8}$	pp.dec ≥ 14.839 ∧ pp.dec ≤ 14.84 ∧ pp.ra ≥ 2.023 ∧ pp.ra ≤ 2.024
AABcl			
1	7.97	81	po.dec[¶] ≥ -42.147 ∧ po.dec ≤ 76.686 ∧ po.ra ≥ 0 ∧ po.ra ≤ 359.821
2	3.53	18.4	pp.dec ≥ -2.7 ∧ pp.dec ≤ 59.6 ∧ pp.ra ≥ 0 ∧ pp.ra ≤ 73
3	3.06	54.3	pp.dec ≥ -4.94 ∧ pp.dec ≤ 91 ∧ pp.ra ≥ 0 ∧ pp.ra ≤ 360
4	2.97	93.35	poa.dec[‖] ≥ -60.572 ∧ poa.dec ≤ 84.98 ∧ poa.ra ≥ 0 ∧ poa.ra ≤ 360
WB			
1	0.6	0.1	pp.dec ≥ 1 ∧ pp.dec ≤ 7.8 ∧ pp.ra ≥ 9.6 ∧ pp.ra ≤ 19.4
2	0.08	0.03	pp.dec ≥ -1.5 ∧ photoprimary.dec ≤ 1.2 ∧ pp.ra ≥ 350.8 ∧ pp.ra ≤ 353.1
3	0.06	0.01	pp.dec ≥ 54.4 ∧ pp.dec ≤ 56.7 ∧ pp.ra ≥ 240.9 ∧ pp.ra ≤ 245
4	0.05	$5.1 \cdot 10^{-6}$	pp.dec ≥ -9.105 ∧ pp.dec ≤ -9.057 ∧ pp.ra ≥ 120.009 ∧ pp.ra ≤ 120.054
FB			
1	27.77		{specobj;specobj.bestobjid}
2	17.15		{photoz; galspecline; specobj; photoz.objid}
3	10.13		{photoobj; photoobj.dec; photoobj.ra}
4	10.11		{phototag; fgetobjfromrecteq; phototag.objid}

5.2.6.4 Influence of Random Sampling on the Clustering Results

Clustering large query logs with the procedure used in this article is time-consuming since one has to (1) extract the query representations, (2) build a proximity matrix, and (3) perform the actual clustering.

Hence, it might be a good idea to cluster a sample of the data. However, so far, it is not clear whether and how sampling influences the result. In particular, it is unclear (1) how the aggregated access areas of clusters using full and sampled data differ, and (2) which clustering results a domain expert finds better. The first answer will be given right away, while the second question is discussed in Section 5.2.6.5.

AABovl clustering. As Figure 5.6 shows, clustering on a sample of the data and the full data yields similar results with AABovl. The differences mainly have to do with cluster ranks, i.e., the position when sorting clusters by their numbers of objects.

Table 5.10: Top clusters of DBSCAN, dataset FullLog

#	**Relative size, %**	**Area coverage, %**	**Aggregated query representation**
AABovl			
1	1.37	0.19	pp.dec ≥ 9.3 ∧ pp.dec ≤ 16.8 ∧ pp.ra[**] ≥ 17.8 ∧ pp.ra ≤ 29
2	0.89	0.18	pp.dec ≥ 0.6 ∧ pp.dec ≤ 8.2 ∧ pp.ra ≥ 9.6 ∧ pp.ra ≤ 19.8
3	0.38	0.03	pp.dec ≥-6.2 ∧ pp.dec ≤ -3.2 ∧ pp.ra ≥ 32.8 ∧ pp.ra ≤ 38.1
4	0.27	0.1	poa.dec ≥ 1.8 ∧ poa.dec ≤ 8.4 ∧ poa.ra ≥ 4 ∧ poa.ra[††] ≤ 10.8
AABcl			
1	4.35	11.27	pp.dec ≥ -11.5 ∧ pp.dec ≤ 59.6 ∧ pp.ra ≥ 0 ∧ pp.ra ≤ 76.4
2	3.06	93.7	poa.dec ≥ -61.551 ∧ poa.dec ≤ 84.98 ∧ poa.ra ≥ 0 ∧ poa.ra ≤ 360
3	1.26	15.76	pp.dec ≥ -24.323 ∧ pp.dec ≤ 84.973 ∧ pp.ra ≥ 279.4 ∧ pp.ra ≤ 360
4	1.16	99.72	as.dec[‡‡] ≥ -90 ∧ as.dec ≤ 87.581 ∧ as.ra ≥ 0.833 ∧ as.ra ≤ 360
FB			
1	27.85		{specobj.bestobjid; specobj}
2	17		{photoz; galspecline; specobj; photoz.objid}
3	10.12		{phototag; fgetobjfromrecteq; phototag.objid}
4	10.07		{photoobj; photoobj.dec; photoobj.ra}

That is why not all clusters occur with both the sampled and the full log: They exist, but not in the top 10.

AABcl clustering. The results with AABcl differ more. Figure 5.6 shows how certain queries "move" from one cluster to another one. It is safe to say that the closeness approach is less robust when it comes to sampling than the overlap approach. Again, the query which has formed a long vertical rectangle and has gone to the second biggest cluster in the sampled data has not disappeared; it just has gone to a less popular cluster, not in the top 10.

FB clustering. As Tables 5.9 and 5.10 indicate, sampling does not change the order of the most popular patterns with FB clustering. We have checked the first 50 popular patterns with sampled and full data and have found only one difference. The ranks change only slightly, by 2 positions at most, and they usually remain the same.

Overall, sampling is useful when clustering an SQL query log. If a query log is huge and requires a lot of time to process, sampling can give way to quick insights. However, AABcl is less robust in this respect.

5.2.6.5 Feedback from a Domain Expert: Clustering Interpretation

As mentioned, a good clustering must be interpretable. Here, this means that each cluster should relate to particular user interest. In astronomy, this means that a cluster may contain several astronomical objects. Still, they all must form a single astronomic category, like "North galactic pole" or "Lockman hole", i.e., represent one research trend. To investigate how successful our clustering has been, and whether it reflects user interests, we have asked a domain expert to inspect our results. He is an astronomer from the Max Planck Institute for Astronomy in Heidelberg, Germany. At the same time, to ease the process of cluster interpretation and ensure a complete representation of the interests of the astronomical community, we have made use of another important astronomical data source, the Simbad astronomical database. We use it as a reference point. Simbad provides information on astronomical objects which have been studied in scientific publications in astronomy. It has 12 tables and contains 9.3 million astronomical objects outside of our solar system and 340 thousand bibliographic references. There are some characteristics common for each astronomical object:

(1) *Basic data*: object types, coordinates and other astronomical features;
(2) *General bibliography for the object*, including references to all published papers from the journals, regularly scanned, currently about 80 titles.

Naturally, two astronomical databases, SkyServer and Simbad, are expected to have a big overlap of the objects they contain. That also holds for attributes like special coordinates, object types etc. However, they are constructed very differently and partly based on different data, so they are quite independent at the same time. With the help of the domain expert, we have mapped our clustering result to the Simbad database. Almost every cluster from our results filters spatial coordinates right ascension "ra" and declination "dec". We have plotted the clusters on the ra-dec plane and have mapped them to the ra-dec density map of astronomical publications.

Of course, one cannot expect a perfect overlap. Not every astronomer looks at the data from SkyServer when writing an article. And vice versa – some data from SkyServer may have been queried for by laymen or high school students, without the publication of a paper. However, both our clusters and Simbad data should reflect hot spots in astronomy. Thus, a relatively high correlation is better as an experiment result, according to our perception. As we have pointed out earlier, our clustering results consist of around 1000 clusters. Identifying user interest in each of them is a daunting task for any domain expert. Such an identification takes our expert 10 minutes on average per cluster, mainly depending on the number of astronomical objects in the cluster. To make manual inspection feasible, we have selected the top 15 clusters from each approach (AABovlFull, AABovlSampled, AABclFull, AABclSampled and WB) which have the most overlap with Simbad data, i.e., clusters which 'repeat' the high-density areas of Simbad.

We assume that these clusters are the most interesting ones for domain experts: There is a high number of publications on the astronomical objects from these particular parts of the sky. Figure 5.7 graphs them together with Simbad data. We have mapped the queries in the various clusters to Simbad data of published papers in the ra-dec plane. A query in the figure is a rectangle which includes admissible ra-dec values. Note that the number near a cluster indicates its' rank: the biggest cluster (in terms of number of queries) has Rank 1. The sixth figure is the pure density map of publications (Simbad). Dark grey areas stand for a high amount of publications; for light grey, the picture is different. One can see that queries from the clusters indeed repeat the distribution of Simbad data: The clusters are located in the grey areas of the Simbad map.

For each cluster, we plot the overlap of the individual query areas on the map with all the Simbad entries. That allows the domain expert to estimate whether the cluster contains one or several astronomic categories of well-studied objects. Having inspected the clusters obtained with DBSCAN, our domain expert has concluded that they are quite different. From his point of view, there are:

(1) Large clusters, each covers more than 3% of the sky or several hundreds or thousands of square degrees. According to our expert, none of them can be associated with a specific scientific goal or type, i.e., there is no corresponding single user interest. In what follows, we refer to these clusters as LwoUI clusters (Large clusters WithOut User Interest). Other large clusters consist of several very small areas, each of which contains a single Simbad entry. For these clusters, our expert has identified a specific user interest. We call these clusters LwUI clusters (Large clusters With User Interest).
(2) Intermediate clusters, each covers less than 3% of the sky. As before, we call them IwUI (Intermediate clusters With User Interest) if they contain user interests and IwoUI (Intermediate clusters WithOut User Interest) otherwise. The domain expert has observed that these clusters have a size so that they likely correspond to a specific scientific goal or type.
(3) Extremely small clusters, which typically consist of several queries referring to the same individual object and cover around 0.01% of the sky. We consequently dub these clusters ESwoUI and ESwUI.

Table 5.11 reveals how many clusters of each category the five approaches (AABovlFull, AABovlSampled, AABclFull, AABclSampled and WB) identify. The table also lists the astronomical objects from the various clusters.

For each scheme, the domain expert has ranked each cluster among the 15 most populated ones according to the probability that it properly covers a region of the sky of particular interest. We have then averaged the grades to rank the five schemes. They take values from one to ten, with ten being the highest interest. AABovlFull has the highest average score, followed by AABclFull, AABovlSampled and AABclSampled. WB is last, see Table 5.11.

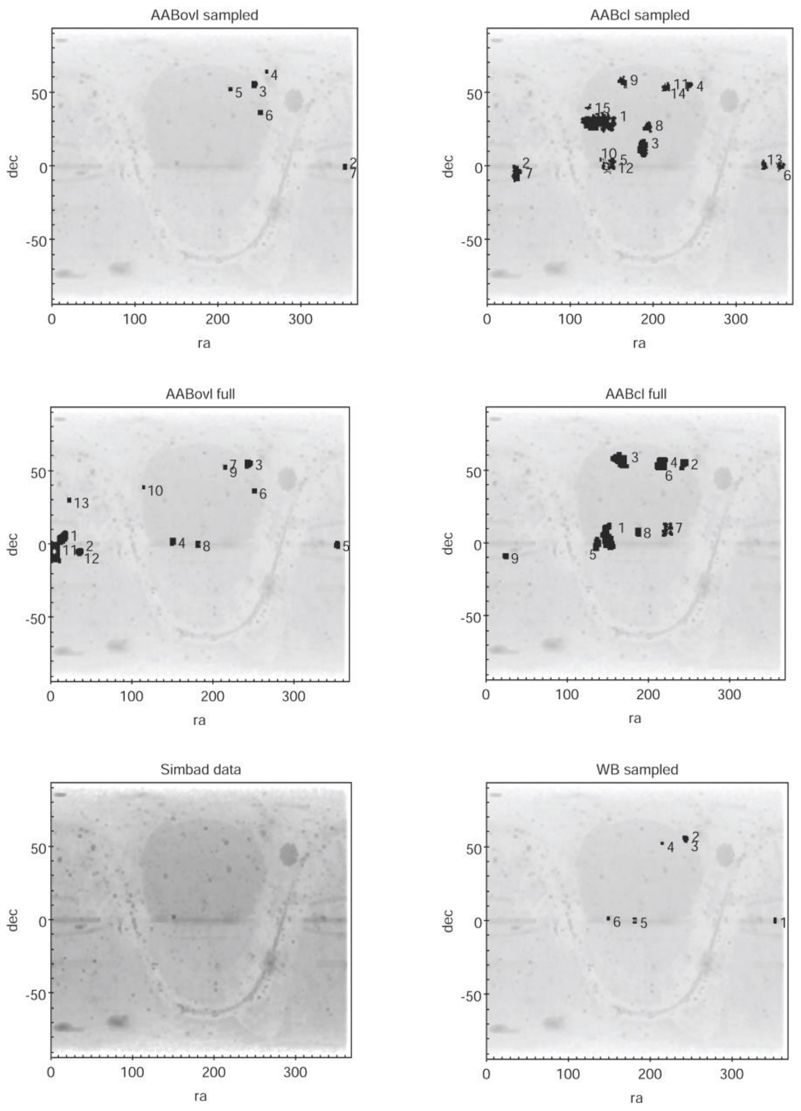

Figure 5.7: Mapping clustering results to Simbad data

One can observe that sampling worsens the clustering results for AABovl. Some clusters disappear, not having enough objects as neighbours. Decreasing minPts value will not always help; the following example shows this:

Example 5.30 Consider the three queries q_1, q_2 and q_3 with the following pairwise distances:

$$D(q_1,q_2)=0.5,\ D(q_1,q_2)=0.8, \text{ and } D(q_1,q_3)=1.\ \textit{eps}=0.9,\ \textit{minPts}=2.$$

All three queries end up in the same cluster. We now sample the log and exclude q_2. Setting *minPts* to 1 does not yield a cluster of queries q_1 and q_3 because they do not overlap. Setting *eps* to 1 does not make any sense because then all queries go to the same cluster.

Based on this, we hypothesize that WB, which also is overlap-based, would have given better results if it had taken place on the full data. We conclude that an analytical calculation of overlap, i.e., AABovl, is useful. It provides sufficient accuracy and is scalable. In consequence, one does not have to do sampling, which bogs down the clustering results.

On the other hand, sampling has helped to obtain better results with the closeness approach, AABcl: Clusters have become smaller and, hence, more focused. Thus, sampling allows identifying user interests than clustering on the original data better.

5.3 Conclusions

Knowing user interests in data space is essential for database owners and domain experts. Clustering the query log can yield interesting insights to this end. In this chapter, we have studied the clustering of SQL query logs. In particular, we have established the design space, i.e., which query representations, which algorithms, which distance measures. Next, we have looked at possible instantiations from the literature systematically and have discussed our expectations for each alternative. We also have proposed new alternatives as well, since the existing proposals have not been fully satisfying. Our new approaches, which we had offered to do away with the weaknesses of existing methods, have turned out to be better in most respects. Finally, we have carried out several studies, one with a domain expert to arrive at a ground truth, a feature which we have not observed in any previous work on analyses of database-query logs. The study with the domain expert, in particular, has revealed the usefulness of clustering when user interests need to be identified.

Our new approach captures query similarity on the data level. Unlike other methods, “witness based” in particular, it scales relatively well with the size of the log.

The next chapter will focus on SQL query recommendation. We plan to leverage our new insights regarding query similarity to find similar user sessions from which

Table 5.11: Results of the study with the domain expert

Approach	**Avg. score**	**LwoUI**	**LwUI**	**IwoUI**	**IwUI**	**ESwoUI**	**ESwUI**	**Astronomical objects**
AABcl Sampled	6.62	0	0	5	10	0	0	Pollux, dwarf galaxy Leo; M77 galaxy; Virgo galaxy cluster: M87 (AGN); ELAIS N1 extragalactic; Cosmos deep field; extragalactic deep field: XMM Deep, CFHT, ESO/VIMOS, Galex; North galactic pole; Lockman hole; Hercules galaxy cluster.
AABovl Sampled	5.53	0	0	10	5	0	0	IC1613; Tadpole galaxy; 3FGL J0008.3+1456; GRB 160410A; Groth-Westphal Strip; M13; Stripe 82.
WB	4.93	0	0	4	7	3	1	IC1613; stripe 86; ELAIS N1 deep field; ClG 0021+0406; Cosmos field; Groth stripe.
AABcl Full	5.4	1	0	5	7	1	1	Cosmos field; ELIAS N1deep field; Lockman hole; Extended Groth Strip; Cross (gravitational lensing).
AABovl Full	7.46	0	0	10	3	0	2	IC1613; Omega Cen (famous giant star), Subaru/XMM-Newton Deep Survey; ELAIS N1; Cosmos deep field; stripe 86; M13; Groth-Westphal Strip; NGC 2419; IRS 1; XMM-LSS deep field; M33.

query suggestions are generated. While SQL query recommendation has already been investigated earlier [Eir+14], [Ali+15], revisiting the topic based on this current study might reveal new insights.

6 Scalable and Data-Aware SQL Query Recommendations

SQL query recommendation suggests an SQL query to a user, based on his submitted queries and the ones of other users stored in a query log. In this chapter, we reused our corresponding work [AB], including figures, tables and algorithms. As we formulated and motivated in Section 1.3, a good SQL query recommendation system (SQL QRS) (1) recommends full and data-aware queries, (2) supports comparison operators and various logical operators, (3) is scalable and has low response times (4) provides recommendations of high quality.

To meet these challenges we have developed DASQR, a data-aware and scalable SQL query recommendation system. Existing approaches introduce several quality metrics to evaluate SQL recommendations. They rely on how an SQL query is represented internally. In our experimental study we not only make use of the existing metrics, but also introduce our own. Our approaches outperform the competing ones according to all metrics both regarding quality and runtime.

6.1 Our Approaches

We now propose new approaches to recommend queries based on a log *US* as a set of user sessions. We offer both collaborative filtering (CF) and content-based (CB) recommendations. To maximize recommendation quality, we also propose hybrid approaches, combining the two principles. As we mentioned in Example 3.10, the existing WB QueRIE hybridization have specific issues. We will pursue a different approach. The objective will always be that hybridization should take place later on the query level, not on the level of query representation, as with [CEP09] or [Eir+14]. The idea is that recommended queries do not mix, but suggestions from alternative approaches appear according to the ranks (priorities) of the respective methods, as we will explain later in Section 6.1.5.

6.1.1 DASQR CF (Collaborative Filtering)

In our context, CF identifies users (user sessions) with similar search preferences (queries). Queries from these user sessions then are recommended. The difficult question is what to recommend if a user has a unique query history. In this case, there are no 'recommendable' queries from the past. With classical recommenders, this is known as the latency problem [SF02]. To overcome it, we propose to construct queries in this case. All in all, we offer two CF schemes, which are independent from each other:

(1) Thick query similarity, no fitting (TkS), (Algorithm 6.1). The method utilizes data-aware query similarities. The similarity of user sessions is obtained by combining similarities of queries contained in them. Finally, we recommend queries from similar user sessions and rank these queries by similarity.

(2) Thin query similarity, thick fitting (TnS) (Algorithm 6.2). We first choose query templates, i.e., the structure of the queries to be recommended. We retrieve queries satisfying these templates from the log *US*. We then transform these queries to a suitable data space – i.e., we construct new filtering conditions.

Section 6.1.2 describes TkS CF, Section 6.1.3 TnS CF.

6.1.2 Thick Query Similarity, no Fitting (TkS)

We first list query similarity measures we deploy or have found in the literature and then present our similarity measure for user sessions.

Algorithmus 6.1: TkS

```
Data: $us_0$, $n$, $US$ – user sessions in a query log
Result: $recQs = \{\}$ – recommended queries, sorted by rank
1 $simUS = $ GetTopNSimilUserSessions($us_0, n, US$)
2 foreach $us \in simUS$ do
3     $qr_i = us$.GetNextQuery($US$)
4     $recQs$.Add($qr_i$);
5 return $recQs$
```

Algorithmus 6.2: TnS

```
Data: $us_0$, $n$, $US$ – user sessions in a query log
Result: $recQs = \{\}$ – recommended queries, sorted by rank
1 $q_n^0 = us_0$.GetLastQuery()
2 $pwp = GoT$.Where(Parrent $= template(q_n^0)$)
3 foreach p $\in pwp$ do
4     $(q_1, q_2) = US$.Where($(template(q_1), template(q_2)) = p$)
5     $qr_i = $ Fitting($(q_1, q_2), q_n^0$)
6     $recQs$.Add($qr_i$);
7 return $recQs$
```

6.1.2.1 Finding Similar Queries

A data-aware query similarity function, needed for TkS, must set q_1 and q_2 from Example 1.6 apart, since they access different data. That is a topic we have studied in Chapter 5. We have proposed two access-area based similarities, AABovl and AABcl. Another data-aware similarity function we utilize is WBBin (binary) similarity (Section 3.3.3.2), which encapsulates witness-based query representation. We have chosen it since in the original experiment it appeared to outperform the result-based alternative WBRes. Thus, we have three alternatives in calculating query similarity for TkS: WBBin, AABovl and AABcl.

6.1.2.2 Identifying Similar User Sessions

For the following discussion the user session we compare the current user session us_0 to is us_1. We take $|us_0| = n+1$, $|us_1| = m+1$. Thus, q^0_{n+1} is an unseen query, and q^1_{m+1} is its possible prediction. Although q^1_{m+1} has been already submitted, while q^0_{n+1} is not, we keep this numbering scheme. So we only compare user sessions upto q^0_n and q^1_m, and there will be no confusion later. A user session similarity function $USS(us_0, us_1)$ quantifies the similarity of two user sessions. We require symmetry of $USS(us_0, us_1)$, i.e., $USS(us_i, us_j) = USS(us_j, us_i)$. Otherwise it is not clear which similarity to use. To test whether the order of queries in session matters, we first propose ordered, then unordered user session similarity (USS) schemes. In the experiments we will compare which approach succeeded.

1. Ordered USS Schemes An ordered $USS(us_0, us_1)$ perceives two user sessions as two sequences of queries. We first introduce the simple measures, where two sessions are compared by pairwise similarities. Later we study existing complex measures which cope with parts of the sequences that do not match.

1.1. Simple Measures With the terminology used here, a simple measure is one that compares two sequences by calculating pairwise query similarities $S(q^0_i, q^1_j)$, $S(q^0_{i+1}, q^1_{j+1})$, etc. and sums them up.

The different lengths issue. When applying simple ordered USS, one faces the issue of different lengths of two sequences, $|us_0| \neq |us_1|$. In this case, some queries, q^0_i or q^1_i, will not be part of a query pair: Let us assume that $n > m$, i.e., $|us_0| > |us_1|$. If we compare q^0_1 with q^1_1, q^0_2 with q^1_2, etc. – i.e., compare two sessions *from the beginning* – there is no q^1_{n+1} to recommend for unseen query q^0_{n+1}. So we compose pairs *from the end* and compare q^0_n with q^1_m, q^0_{n-1} with q^1_{m-1}, etc. Thus, q^1_{m+1} is a recommendation for q^0_{n+1}. Yet we need to decide what to do when a query is not part of a pair, having in mind that $USS(us_0, us_1)$ is symmetric.

Example 6.1 Let us consider two user sessions: $us_0 = (q_1^0, q_2^0)$, $us_1 = (q_1^1, q_2^1, q_3^1, q_4^1)$; $S(q_1^0, q_1^1) = 0, S(q_1^0, q_2^1) = 0, S(q_1^0, q_3^1) = 1$. q_2^0 is an unseen query, and q_4^1 is its possible prediction. If we take only $S(q_1^0, q_3^1) = 1$ into account when calculating $USS(us_0, us_1)$, we overestimate the similarity of us_0 and us_1: They are not identical.

Instead, we 'complement" the shortest user session us_0 from the beginning, so $|us_0| = |us_1| = 4$ in the example, and set $S(q_{-2}^0, q_1^1) = 0$ and $S(q_{-1}^0, q_2^1) = 0$. Now all three query similarities contribute to $USS(us_0, us_1)$, and the similarity function is symmetric.

The weighting schemes. We hypothesize that the importance of query similarity increases from the beginning of the sessions to the end. In other words, it may not be so vital for a prediction whether the first queries of two sessions are different, but it is much more significant if the last SQL requests are not alike. To test this hypothesis, we introduce weighting schemes of ordered user session similarity (USS WS). Here, the similarity of the two queries is multiplied with the position of these two queries in the corresponding user sessions. Having said this, we now present the weighting schemes:

(1) Flat scheme, where all query similarities $S(q_i^0, q_i^1)$ have the same impact on $USS(us_0, us_1)$:

$$USS_{flat}(us_0, us_1) = \frac{1}{k} \cdot \sum_{i=1}^{i=k} S(q_{i-l_0}^0, q_{i-l_1}^1) \tag{6.1}$$

Here, $k = max(|us_0|, |us_1|)$, $l_0 = k - |us_0|$, $l_1 = k - |us_1|$.

(2) Geometric progression (GP) scheme, where similarities of queries close to the end of the session have a high weight:

$$USS_{GP}(us_0, us_1) = \sum_{i=1}^{i=k} S(q_{i-l_0}^0, q_{i-l_1}^1) \cdot a_1 \cdot r^{i-1} \tag{6.2}$$

Here, $r < 1$. a_1 and r are chosen so that the sum to infinity of a GP $a_i = a_1 \cdot r^{i-1}$ is 1: $S_\infty = a_1/(1-r) = 1$.

(3) Only Last Query (OLQ) scheme, where one considers only the last queries of two user sessions:

$$USS_{OLQ}(us_0, us_1) = S(q_{|us_0|}^0, q_{|us_1|}^1) \tag{6.3}$$

1.2. Complex measures Simple ordered $USS(us_0, us_1)$ does not cope with parts of the sequences that do not match. There are dynamic distance measures designed to overcome this. The Smith-Waterman algorithm (SWA) [SW+81] searches for similar subsequences in two sequences. Another algorithm, dynamic time warping (DTW) [Sak+90], quantifies the similarity of two temporal sequences which may vary in speed. Both algorithms have a complexity of $O(m \cdot n)$, where m and n are the lengths of the sequences. We will test if complex measures outperform simple ones in recommendation quality.

2. Unordered USS Scheme. The unordered USS scheme perceives a user session as a set of queries. According to it, $USS_{unord}(us_0, us_1)$ is the sum of pairwise similarities, no order is considered:

$$USS_{unord}(us_0, us_1) = \frac{1}{|us_0| \cdot |us_1|} \cdot \sum_{i=1,j=1}^{i=|us_0|,j=|us_1|} S(q_i^0, q_j^1) \tag{6.4}$$

6.1.2.3 Recommending Queries

The final step, when providing recommendations for a given user session, picks the next queries from similar sessions (Algorithm 6.1, Line 3).

Note that $USS(us_0, us_1) = 0$ occurs quite frequently. Next, there may not be sufficiently many similar user sessions: $|simUS| < nr$ where $simUS$ consists of only non-zero similarities.

6.1.2.4 Limitation of TkS

TkS limits the recommendations to the queries in the log. Naturally, if the information need of a user does not correspond to any query in the log, recommendations are not generated. The approach proposed next, TnS, aims to overcome this problem: We construct a recommended query instead of getting it from a log.

6.1.3 Thin Query Similarity, thick Fitting (TnS)

TnS is another new technique to suggest queries. It first finds FB queries (templates) to recommend. Then these templates are instantiated with filtering conditions and become recommended queries. In what follows, we describe this process step by step.

6.1.3.1 Recommending Templates

FB QueRIE template recommendation (FB QueRIE TR) Our literature review has featured an approach recommending templates, FB QueRIE [Eir+14]. It relies on feature similarity and computes the prediction summary S_0^{pred}, a vector of features. The method recommends the templates most similar to S_0^{pred}.

Graph-based template recommendation (GBTR) We wonder if there are simpler and more effective ways to suggest templates. In line with classical recommender systems, we use Markov chains, similarly to recommendations of query fragments in SnipSuggest [Kho+10]. Algorithm 6.3 illustrates the principle. The approach makes use of our previous work in Chapter 4. In a nutshell, the method analyzes patterns (Definition 4.3) in a query log and suggest the most popular continuation to a given query template (Definition 4.5). Algorithm 6.3 illustrates the principle.

Definition 6.2 A graph of templates GoT is a directed graph. A vertex is a template. The weight of an arrow $weight(t_i, t_j)$ is the probability of having t_j immediately after t_i in a query log US.

To build a GoT, one first extracts features from queries and assigns the template to each SQL request. See Algorithm 6.3. Line 7 sets the probabilities:

$$GoT.\text{weight}(t_i, t_j) = \frac{GoT.\text{weight}(t_i, t_j)}{\sum_{a \in GoT.\text{TailsOf}(t_i)} GoT.\text{weight}(t_i, t_a)} \tag{6.5}$$

As intermediate result, one gets a set of patterns with corresponding probabilities $pwp = GoT.\text{Where}(\text{Head} = template(q_n^0))$. Note that all patterns from pwp start with $template(q_n^0)$. To derive pwp for FB QueRIE recommendation, we set p_i depending on how similar S_0^{pred} is to t_i:

$$p_i = \frac{S_{FB}(t_i, S_0^{pred})}{\sum_{t \in T} S_{FB}(t, S_0^{pred})} \tag{6.6}$$

Here, T is the set of all templates present in the log US. S_{FB} is either Jaccard or cosine similarity, depending on the FB weighting scheme used. Again, the sum of all p_i's is 1. Having $p_1 \geq p_2 \geq \dots \geq p_k$, the index i of p_i in pwp ranks the recommendation.

6.1.3.2 Fitting

Recall that one wants to generate nr query recommendations. The fitting step instantiates filtering conditions of recommended queries $\{qr_1, \dots qr_{nr}\}$. It takes pwp as input. To get $\{qr_1, \dots qr_{nr}\}$, we first retrieve $nr_i = i \cdot trunc(nr \cdot p_i)$ instances of each pattern $(template(q_n^0), t_i)$ present in pwp. Each instance (q_1, q_2) of a pattern then must pass the fitting step. Algorithm 6.4 specifies the steps. The purpose of fitting is to generate recommended query qr_i. Note that (q_n^0, qr_i) and (q_1, q_2) are instances of the same pattern.

The fitting procedure $Fitting((q_1, q_2), q_n^0)$ described in Algorithm 6.4 modifies query q_n^0 depending on the difference of the access areas of queries q_1 and q_2. Each attribute in filtering conditions of q_2, $Ints(q_2)$, is processed independently (Line 1). To generate filtering conditions of Attribute a in qr_i ($Ints(qr_i)$) we need to have Attribute a in $Ints(q_n^0)$ and in $Ints(q_1)$ or a *related* attribute:

Definition 6.3 Attributes a' and a are related iff $(a'.name = a.name) \wedge (a'.type = a.type)$.

The definition refers to the case of attributes in different tables. If a' and a belong to the same table, $a' = a$. Thus, a is related to itself. One could extend the definition of related attributes by considering, for instance, the semantic of the tables. This, however, would require domain knowledge separately for each individual database. In the use case of our study, the Sky Server database, it would mean to examine over 180 tables and views with around 50-100 attributes in each table. By introducing a simple notion of related attributes, we wanted to study how far one gets in the most prominent cases and to avoid any manual intervention.

Setting the filtering conditions of a depends on the attribute type, which may be ordinal or nominal.

1. Fitting for an ordinal attribute For an ordinal attribute a, fitting preserves the transformation of an access area from q_1 to q_2 (Algorithm 6.4, line 4). More specifically, to set an interval ar_i (an interval of attribute a in qr_i), fitting adjusts two values:

(1) The shift ar_{shift} is a distance between two centers of intervals ar_i and a'_n. a'_n is an interval of attribute a' in q_n^0.

(2) The width of an interval $width(ar_i)$ indicates how big is the access area of a query qr_i on attribute a.

Algorithmus 6.3: BuildGoT

```
Data: $US$ – user sessions in a query log
Result: $GoT = \{\}$ – a directed acyclic graph of tempaltes
1 foreach us ∈ US do
2     $prevQ = null$
3     foreach q ∈ us do
4         if $prevQ \neq null$ then
5             $GoT$.weight($template(prevQ), template(q)$)++
6             $prevQ = q$
7 $GoT$.SetProbabilities()
8 return $GoT$
```

In essence, we keep the proportions of the query intervals of the pair (q_1, q_2) and translate them to intervals of the pair (q_n^0, qr_i):

$$width(ar_i) = width(a_2) \cdot width(a'_n)/width(a'_1) \tag{6.7}$$

$$ar_{shift} = a_{shift} \cdot width(a'_n)/width(a'_1) \tag{6.8}$$

Algorithmus 6.4: Fitting

```
Data: $(q_1, q_2)$, $q_n^0$
Result: $qr_i$ – a recommended query
1  foreach $a \in Ints(q_2)$ do
2      $a' = $ GetRelatedAttribute$(a)$
3      if $a'$ is ordinal then
4          $qr_i$.SetIntervalForAttribute$(a', q_1, q_2, q_n^0)$
5      else
6          if $S_{ovl}(q_1, q_n^0).a \neq 1$ then
7              $(q_1, q_2) = US$.Where$(S_{ovl}(q_1, q_n^0).a = 1)$
8              if $(q_1, q_2) \neq null$ then
9                  $qr_i$.CopyValuesForAttribute$(a', q_2)$
10             else
11                 $(q_1, q_2) = US$.Where$(S_{ovl}(q_1, q_2).a = 1)$
12                 if $(q_1, q_2) \neq null$ then
13                     $qr_i$.CopyValuesForAttribute$(a', q_n^0)$
14 return $qr_i$
```

Example 6.4 q_1, q_2 and q_n^0 are as follows:

q_1 `SELECT name, population FROM Countries Cs WHERE Cs.latitude BETWEEN 30 AND 50`

q_2 `SELECT name, population FROM Cities C WHERE C.latitude BETWEEN 40 AND 50`

q_n^0 `SELECT name, population FROM Countries Cs WHERE Cs.latitude BETWEEN 30 AND 60`

Attributes Cs.latitude and C.latitude are related by Definition 6.3. We now derive the filtering conditions of qr_i:

width: $width([C.latitude]r_i) = (50-40) \cdot (60-30)/(50-30) = 15$

shift: $a_{shift} = 45-40 = 5; ar_{shift} = 5 \cdot (60-30)/(50-30) = 7.5$

Hence, qr_i is: `SELECT name, population FROM Cities C WHERE C.latitude BETWEEN 45 AND 60`

2. Fitting for a nominal attribute With nominal attributes, one cannot shift values in the same way as with ordinal attributes. Hence, we try to get the pair (q_1, q_2) from a query log US so that $S_{ovl}(q_1, q_n^0).a = 1$ (Algorithm 6.4, Line 7). Then we can copy filtering conditions for a of q_2 to qr_i (Line 9). If there is no such pair, we see if there is a query in US where $S_{ovl}(q_1, q_2).a = 1$. If so, we copy filtering conditions for a of query q_n^0 to query qr_i (Line 13). If not, TnS cannot provide a recommendation.

6.1.3.3 Limitation of TnS

The ranked list of recommendations $(q'_1, \dots q'_{nr})$ can also contain fewer queries than nr. We foresee the following situations when TnS cannot produce recommendations:

(1) All instances of template $t = template(q_n^0)$ belong to the current user session. In this case, graph-based template recommendation GBTR cannot suggest a template.

(2) There are no related attributes in interests of q_1 and q_2.

We will study how often this occurs with a real-world log.

6.1.4 DASQR CB (Content-Based)

The content-based recommendation suggests queries similar to the ones a user has issued. Here we use three data-aware query similarity functions, WBBin, AABovl and AABcl. That gives way to three CB recommendation schemes: CB WB, CB AABovl and CB AABcl.

6.1.5 Hybrid Approaches

With new hybrid approaches, we intend to maximize the utility of our methods: CF TkS, CF TnS and CB. We propose to go away from a hybridization on a query representation level (as in WB QueRIE). We hypothesize that each possible recommendation carries one and only one user interest. We will validate this later on by comparing our hybrid approaches to those from WB QueRIE, where queries (their tuples) are blended.

Our procedure is the following one: CF Tks, CF TnS and CB all take place, each approach trying to come up with nr recommendations. For a particular hybridization scheme, we establish an order of the algorithms. To illustrate, think of (1) CF Tks, (2) CF TnS, (3) CB. If due to its limitations, CF TkS does not produce (enough) recommendations, we resort to suggestions from CF TnS. If the combined number of recommendation is not enough, smaller than nr, we take suggestions from CB. So the results of each hybrid method will be at least as good as the algorithm that is first.

This method, albeit simple, is in line with the design objective of having only one user interest within a suggested query. In the experimental section, we will study if it outperforms the hybridization of WB QueRIE.

6.1.6 Summary

Figure 6.1 summarizes the various DASQR methods. '+' states that the methods of this part of the hierarchy do not exclude each other, but can be applied together. For example, TnS performs both recommending templates and fitting. In contrast, a subhierarchy without '+' means that one must choose one method from it. For instance, one can use only one query similarity at a time.

All the approaches we propose satisfy the requirements from Section 2.5:

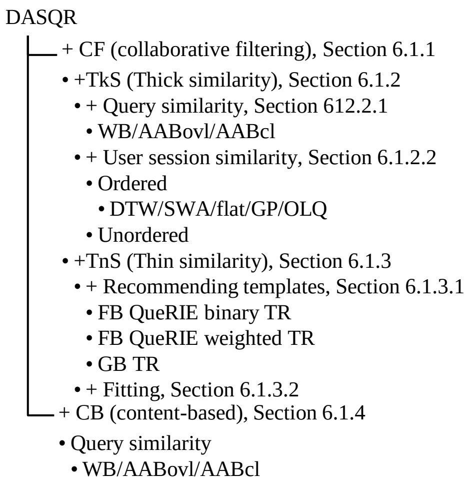
DASQR
+ CF (collaborative filtering), Section 6.1.1
• +TkS (Thick similarity), Section 6.1.2
• + Query similarity, Section 612.2.1
• WB/AABovl/AABcl
• + User session similarity, Section 6.1.2.2
• Ordered
• DTW/SWA/flat/GP/OLQ
• Unordered
• +TnS (Thin similarity), Section 6.1.3
• + Recommending templates, Section 6.1.3.1
• FB QueRIE binary TR
• FB QueRIE weighted TR
• GB TR
• + Fitting, Section 6.1.3.2
+ CB (content-based), Section 6.1.4
• Query similarity
• WB/AABovl/AABcl

Figure 6.1: The design decisions of DASQR

(1) They recommend full and data-aware queries: Our new approaches either search for queries/user sessions that are similar in the given data space (CF TnS in Section 6.1.2, CB in Section 6.1.4), or they construct filtering conditions of the recommended query (CF TnS in Section 6.1.3).

(2) DASQR recommends queries with comparison operators, including range queries. When we deploy the AAB query-similarity measure, our methods inherit the AAB restriction of not having an arithmetic operation in filtering conditions [Arz+19]. Having arithmetic operations in SQL is bad practice anyway. [*]

(3) Scalability of DASQR depends in many ways on the query similarity function in use. An earlier study of similarity functions reveals that WB tends to be slower than AAB [Arz+19]. However, even with WB similarity, DASQR is faster than WB QueRIE, which uses the WB representation as well.

[*] See https://www.red-gate.com/products/sql-development/sql-prompt/entrypage/code-smells?v=1 under the heading "Including complex conditionals in the WHERE clause".

6.2 Experiments

This section features experiments. We proceed as follows:

(1) Define quality metrics for SQL query recommendations. We base our metrics on corresponding SQL query similarity metrics, WB, AABcl, and AABovl.

(2) Compare the quality of the various approaches on the SkyServer query log according to those metrics;

(3) Measure the response times of approaches. Conclude which ones are best overall in terms of speed and accuracy.

In Section 3.4.3, we have reviewed techniques to recommend SQL queries and their compliance with the requirements. None of them meets all requirements. WB QueRIE, however, has only one downside – scalability. So one can at least select a relatively small data set to bring down runtime problems when testing it. We have chosen WB QueRIE (Bin and Weig) as our reference point. The other approaches cannot be compared with ours experimentally since they either (1) do not provide full and data aware queries [YPS09], [Kho+10], [Eir+14] or (2) do not support range queries [Ali+15] or (3) run on a database instance and rely on query plans [KBGM09].

6.2.1 Experiment Setup

We evaluate the predictions made by all DASQR approaches and competitors offline, i.e., without users [RRS11b]. Doing online experiments is a huge task, which is out of the scope of this thesis.

6.2.1.1 The Query Log

Sloan Digital Sky Survey (SDSS, or SkyServer), is a project targeting at a map of the universe. Professionals and hobby astronomers are participating. The data is available online. There also is an SQL command line to the SDSS database server, allowing to compose queries of any complexity.

According to [GS09], choosing the data for offline experiments should be done in a way that it would be most similar to the online application. Thus, the log must be cleaned from sliding window search (SWS) queries issued by robots.[†]. We have utilized the same log (cleaned from SWS), as for clustering (Section 5.2.2.2), but we had to exclude user sessions, where not all queries have their corresponding query representations (WB, FB and AAB). We have made the log publicly available[‡].

[†]We have discussed the procedure in Chapter 4

[‡]https://anonymous.4open.science/r/5f6d2410-83b8-463a-8113-f517d9a49e6e/

The log (FullLog) is of 79,151 queries. It touches 179,611,275 tuples in the SkyServer database. This log is too big to be processed by WB QueRIE. To compare with WB QueRIE to some extent and to study the influence of sampling on recommendation quality, we have randomly chosen 1/10 of the user sessions from FullLog to form what we call SampledLog.

6.2.1.2 Algorithms to Compare

Have a look at Figure 6.1 again. The variety of DASQR algorithms is wide due to several design options on each level. The first distinction is on the general recommendation approach: CF or CB. For CF, one goes with TkS or TnS. When it comes to TkS, one must decide on user sessions and query similarities. With TnS the options lie in choosing an algorithm to recommend templates.

Table 6.2 lists the DASQR algorithms, which have performed best in the experiments. The first part of the name of the algorithm points to the query similarity function, if applicable. The second part indicates which user session similarity is used. Then there are identifiers of additional parameters if needed. For instance, "CF AABovlGp03" is a CF TkS approach which utilizes AABovl query similarity and the geometric progression weighting scheme with $r = 0.3$ when it comes to computing user sessions similarity. For hybrid methods, we list the algorithms applied in their order. For example, H (1) + (8) + (9) from Table 6.2 takes the recommendations from (1). If this is not enough, it continues with (8), then with (9).

6.2.2 Evaluation Protocol

We perform the experiments offline, predicting the next query for a given user session. We apply 10-fold cross-validation, dividing users sessions into ten groups. Each time nine of them serve as the training set, one as the input set. We treat a user session $us_0 = (q_1^0, \dots q_n^0)$ as n user sessions $us_0[1;1], \dots, us_0[1;n]$, trying to predict queries from q_2 to q_n. We do the same with the other user sessions: We perceive $us_1 = (q_1^i, \dots, qm+1^i)$ as a set of m user sessions $\{us_i[1;2], \dots, us_i[1;m+1]\}$. $us_i[1;k]$ is obtained from the original us_i by taking only the first k queries, $k \leq m+1$. Thus, any query from q_2^i to q_{m+1}^i can be a possible prediction.

6.2.3 Evaluation Metrics

To evaluate predictions, one often uses precision and recall, defined as follows for singleton items:

$$P = |tp| / (|tp| + |fp|); R = |tp| / (|tp| + |fn|), \tag{6.9}$$

where $|tp|$ is the number of true positive items (items both recommended and of interest), $|fp|$ is the number of false positive items (recommended, but not of interest) and $|fn|$ is the number of items not recommended, but of interest (false negatives).

6.2.3.1 Data-aware metrics

As mentioned, SQL queries are different from "items" like movies or tangible goods. The literature explains that quality metrics for SQL query recommendation are specific to the query representation and query similarity function. For example, with WB QueRIE [CEP09], precision and recall are defined based on tuples:

$$P_{WB} = \left|\tau_{Q_U} \cap \tau_{Q_R}\right| / \left|\tau_{Q_R}\right|, \quad R_{WB} = \left|\tau_{Q_U} \cap \tau_{Q_R}\right| / \left|\tau_{Q_U}\right| \tag{6.10}$$

where τ_{Q_U} stands for the witnesses of the unseen query q_U and τ_{Q_R} for the witnesses of the recommended query q_R. The accuracy metric in WB should then be:

$$A_{WB} = \left|\tau_{Q_U} \cap \tau_{Q_R}\right| / \left|\tau_{Q_U} \cup \tau_{Q_R}\right| \tag{6.11}$$

Formula (6.11) is an instantiation of Jaccard similarity. The similarity is the relative size of the intersection of two query results. For precision, the denominator changes from the union of the tuples ($\tau_{Q_U} \cup \tau_{Q_R}$) to the tuples of the recommended query alone (τ_{Q_R}), for recall to the tuples of the unseen query (τ_{Q_U}). Following this practice, we redefine precision and recall from AABovl and AABcl similarities.

AABovl Precision and Recall Let q_U denote an unseen query and q_R a recommended one. Transforming AABovl similarity for an ordinal attribute (Formula (5.4)) to precision and recall, we get:

$$P_{AABovl}.a^{OA} = comWidth(a^{OA}_{U.1}, a^{OA}_{R.1}) / width(a^{OA}_{R.1}) \tag{6.12}$$

$$R_{AABovl}.a^{OA} = comWidth(a^{OA}_{U.1}, a^{OA}_{R.1}) / width(a^{OA}_{U.1}) \tag{6.13}$$

For a nominal attribute, the formulas are identical for the closeness and the overlap approach:

$$P_{AABcl}.a^{NA} = P_{AABovl}.a^{NA} = \left|A^{NA}_U \cap A^{NA}_R\right| / \left|A^{NA}_U\right| \tag{6.14}$$

$$R_{AABcl}.a^{NA} = R_{AABovl}.a^{NA} = \left|A^{NA}_U \cap A^{NA}_R\right| / \left|A^{NA}_R\right| \tag{6.15}$$

AABcl Precision and Recall With closeness precision and recall, things are more complicated. Obviously, we cannot change Formula (5.2) of AABcl similarity by replacing the denominator with $width(a^{OA}_{R.j}))$ or $width(a^{OA}_{U.i})$: We would get $P_{AABcl} > 1$ and $R_{AABcl} > 1$ in some cases. Instead, we perceive "the average interval" $a^{OA}_{avg.ij}$ of $a^{OA}_{U.i}$ and $a^{OA}_{R.j}$ as one whose width is an average of $width(a^{OA}_{U.i})$ and $width(a^{OA}_{R.j})$ and whose center is the "mass center" of $a^{OA}_{U.i}$ and $a^{OA}_{R.j}$. Consequently, the precision and recall is a closeness similarity (Formula (5.10)) of this interval and $a^{OA}_{U.i}$ or $a^{OA}_{R.j}$. The attribute-wise Precision and Recall then are:

$$P_{AABcl}.a^{OA} = S_{cl}(a^{OA}_{U}, a^{OA}_{avg}) \tag{6.16}$$

$$R_{AABcl}.a^{OA} = S_{cl}(a^{OA}_{R}, a^{OA}_{avg}) \tag{6.17}$$

Overall Precision and Recall of AABcl and AABovl.

The final formulas result from changing accuracy to precision and recall:

$$P_{attrFull}(q_U, q_R) = min(P_{attrExcl}(q_U, q_R), P_{attrCom}(q_U, q_R)) \tag{6.18}$$

$$P_{attrCom}(q_U, q_R) = \min_{i=1,\dots,|comInts(q_U,q_R)|} P(q_U, q_R).a^i \tag{6.19}$$

$$P_{attrExcl}(q_U, q_R) = \min_{i=1,\dots,|exclInts(q_U,q_R)|} P_{ovl}(q_U, q_R).a^i \tag{6.20}$$

$$P(q_U, q_R) = reductCoeff_{table} \cdot P_{attrFull}(q_U, q_R) \tag{6.21}$$

$$R_{attrFull}(q_U, q_R) = min(R_{attrExcl}(q_U, q_R), R_{attrCom}(q_U, q_R)) \tag{6.22}$$

$$R_{attrCom}(q_U, q_R) = \min_{i=1,\dots,|comInts(q_U,q_R)|} R(q_U, q_R).a^i \tag{6.23}$$

$$R_{attrExcl}(q_U, q_R) = \min_{i=1,\dots,|exclInts(q_U,q_R)|} R_{ovl}(q_U, q_R).a^i \tag{6.24}$$

$$R(q_U, q_R) = reductCoeff_{table} \cdot R_{attrFull}(q_U, q_R) \tag{6.25}$$

So we now have definitions for WB, AABovl and AABcl precision and recall metrics.

6.2.3.2 Metrics for query templates

We also want to evaluate the quality of giving template recommendations of FB QueRIE and our graph-based approach (GBTR) for TnS (Section 6.1.3). To calculate Precision and Recall for query templates, we have used the formulas from the original FB QueRIE article.

6.2.4 Experimental Evaluation

In this section, we present the results of our experiments. First, we report on the quality of recommending templates for TnS. Then we report on characteristics of DASQR and WBQueRIE.

6.2.4.1 Quality of Recommending Templates

As we have said earlier (see Section 6.1.3.1), TnS works with any method which recommends templates. We have introduced a new graph-based approach (GBTR) and described the existing ones, FB QueRIE binary and weighted. We have obtained template recommendations of all three approaches. Table 6.1 shows the average maximum recall and the subsequent precision characteristic. Since GBTR appears to be best, we have used it for TnS query recommendations.

As pointed out in Section 6.1.3.3 (Case (1)), GBTR cannot recommend a template if all instances of template $t = \mathit{template}(q_n^0)$ belong to the current user session. Due to 10-fold cross-validation, one cannot provide a template recommendation after template t iff all instances of t belong to the test set. That occurred in 1.7 % of the cases for FullLog and in 9.3 % of the cases for SampledLog.

6.2.4.2 Quality of Recommending SQL Queries

Tables 6.2 and 6.3 report average maximum recall and average precision on maximum recall on FullLog and SampledLog. We show only the best pure and hybrid algorithms. To get the values, we first take the maximum recall for each unseen query and obtain the corresponding precision. We then report averages of these precision and recall values. For each algorithm, we have set $nr = 10$. Figures 6.2 and 6.3 feature with respective inverse CDFs.

We have performed experiments where only five, three and one recommendations are get as well. To save space, we do not present the outcomes here. The results, as expected, get worse with fewer recommendations. The order of the algorithms with respect to prediction quality changes only slightly.

Table 6.1: Average maximum Recall (R) and Average Precision on Maximum Recall (P) for recommending templates

Value / **Method**	P_{FB} **Full**	R_{FB} **Full**	P_{FB} **Sampled**	R_{FB} **Sampled**
FB QueRIE binary	0.368	0.317	0.349	0.312
FB QueRIE weighted	0.289	0.275	0.273	0.278
GBTR	**0.944**	**0.873**	**0.843**	**0.789**

Table 6.2: Average maximum Recall (R) (in percent) and Average Precision on Maximum Recall (P) (in percent), dataset - Full

Method	P_{ovl}	R_{ovl}	P_{cl}	R_{cl}	P_{WB}	R_{WB}
(1) CF AABovlGp03	26.6	21.6	25.5	28.5	20.8	19.2
(2) CF AABovlOLQ	27.7	*23.2*	26.6	28.8	*23.4*	*21.7*
(3) CF AABovlDTW	23.4	18.9	22.5	25.1	19.2	18
(4) CF AABclOLQ	*28.1*	23.1	*29.5*	*35.1*	24	22
(5) CF AABclDTW	20.7	20.1	22.1	22.2	20.6	19.3
(6) CF WBOLQ	12.4	12.1	14.1	14.2	22.2	20.4
(7) CF WBDTW	12.8	12.2	14.7	15.1	20.6	19.2
(8) CF TnS	*14.9*	*14.9*	*15.1*	*15.7*	*18.8*	*26.5*
(9) CF CB AABcl	*19.8*	*12.9*	*21.5*	*30.1*	*16.5*	*14.4*
(10) (1) + (8) + (9)	28.9	23.9	28.9	32.6	23.4	22.7
(11) (2) + (8) + (9)	**33.5**	**28.7**	**33.2**	36.9	**30.8**	31.8
(12) (2) + (9) + 8)	31.9	26	31.9	**38.5**	25.8	24.2
(13) (8) + (1) + (9)	23.3	20.3	15.8	20.1	19.8	7.1
(14) (8) + (2) + (9)	22.5	19.8	23.2	26.4	26.1	**32.6**
(15) (9) + (1) + (8)	23.3	20.3	23.3	26	26.1	32.4
(16) WB QueRIEWeig	-	-	-	-	-	-
(17) WB QueRIEBin	-	-	-	-	7	-
(18) Best possible	77.5	63.7	72.6	88.5	39.8	39.7
(19) Baseline	0.4	0.4	1	1.1	0.3	0.3

Table 6.3: Average maximum Recall (R) (in percent) and Average Precision on Maximum Recall (P) (in percent), dataset - Sampled

Method	P_{ovl}	R_{ovl}	P_{cl}	R_{cl}	P_{WB}	R_{WB}
(1) CF AABovlGp03	9.2	8.3	9.2	9.9	9.2	9.3
(2) CF AABovlOLQ	*9.5*	*8.4*	8.8	9.1	8.9	8.9
(3) CF AABovlDTW	9.1	8	9.2	10.3	9.6	9.6
(4) CF AABclOLQ	9.5	*8.4*	*11.2*	15.9	9.3	9.3
(5) CF AABclDTW	8.1	7	10.4	*16.1*	8.8	8.5
(6) CF WBOLQ	4.8	4.4	4.9	5	8.4	8.6
(7) CF WBDTW	5.7	5.2	6.2	6.7	*10.3*	10
(8) CF TnS	*12.5*	*12*	*11.7*	*12.3*	*16.6*	*2.8*
(9) CF CB AABcl	*7.2*	*5.4*	*10.7*	*19.9*	*7.7*	*7.4*
(10) (1) + (8) + (9)	17.4	15.9	18.7	23.1	18.9	11.8
(11) (2) + (8) + (9)	**19.5**	**18**	**23**	26.9	**22.3**	**12.1**
(12) (2) + (9) + 8)	16.1	14.4	19.6	**28.3**	20	11.9
(13) (8) + (1) + (9)	15	14	15.8	20.1	19.8	7.1
(14) (8) + (2) + (9)	15.2	14.1	16.4	20.9	17.6	7.2
(15) (9) + (1) + (8)	16.6	14.2	19.7	28.3	16.2	9.6
(16) WB QueRIEWeig	7.2	6.8	8.5	8.9	7.2	6.9
(17) WB QueRIEBin	7.1	6.7	8.3	8.8	7	6.8
(18) Best possible	30.6	27.8	46.9	54.9	18.8	19.2
(19) Baseline	0.2	0.2	0.4	0.7	0.1	0.2

We now explain the last two columns in Tables 6.2 and 6.3. 'Best possible' is as follows: For each unseen query q_U, we choose the query q' from the corresponding proximity matrix (WBBin, AABovl or AABcl) with maximal recall and retrieve precision and recall values for it. Theoretically, TnS and hybrid methods can surpass it, since their recommendations are not limited to queries from the log. 'Baseline' recommends a random query from the log.

Low precision and recall values when recommending data-aware queries are not unexpected. First, as 'Best possible' values show, 1 is not achievable. Second, the baseline values indicate that the performance of the methods is not a result of chance.

In what follows, we describe the effects of various design decisions (see Figure 6.1), report on general patterns of quality metrics and compare our approaches with the reference points:

(1) *General patterns of quality metrics.* Results are stable, irrespective of the query representation the quality metric is built upon. For instance, almost all metrics identify CF AABclOLQ as the best CF algorithm. The pair-wise Spearman correlations of average maximum precision and corresponding recall are: $\rho_s(P_{ovl}, P_{cl}) = 0.95$, $\rho_s(R_{ovl}, R_{cl}) = 0.76$, $\rho_s(P_{ovl}, P_{WB}) = 0.63$, $\rho_s(R_{ovl}, R_{WB}) = 0.38$, $\rho_s(P_{cl}, P_{WB}) = 0.57$,$\rho_s(R_{cl}, R_{WB}) = 0.2$. AABovl and AABcl metrics correlate strongly since they both use the AAB query representation. WB correlates more with AABovl because these two metrics target at the overlap in the data space. AABovl does this analytically, WB literally.

(2) *Comparison with reference points.* Our best hybrid algorithms are much better than WB QueRIE. Considering their low runtimes in addition (see Table 6.4), they are clear winners.

(3) *WB QueRIE hybridization vs our hybridization.* We compare the results of WB QueRIE to the ones of any of our hybrid algorithms, which use WB representation, on sampled data (see Table 6.3). Our hybrid algorithms performed around two times better regarding Precision and Recall. Thus, our objective of not mixing queries has turned out to be meaningful.

(4) *Applicability of CB.* CB methods performed moderately well. Since CB recommends items similar to the ones a user has already asked for, this indicates that, as in e-commerce, a database user is interested in queries similar to ones he has just submitted. We recommend utilizing CB query recommendation in combination with CF.

(5) *TnS and TkS.* TnS is worse than TkS with the FullLog. With SampledLog, TkS gets worse much faster than TnS. That is because TkS requires similar queries, whereas TnS does not. In other words, TnS is less sensitive to scaling the size of the log down.

(6) *TnS performance.* As explained in Section 6.1.3.3 (Case (2)), if the last query of a current session and a predicted query do not have related attributes in their filtering conditions, TnS cannot recommend a query. That has happened in 18.1 % of the cases for FullLog and in 16.9 % of the cases for SampledLog.

(7) *Unordered vs ordered user session similarity.* Taking the order of queries into account when calculating session similarity is important: 'Ordered' approaches always perform better.

(8) *Complex ordered USS.* Complex measures of computing ordered user session similarity (DWT and SWA, see Section 6.1.2.2) are not clear winners. With SampledLog, CF WBDTW has been best only regarding P_{WB} and R_{WB}, whereas user session similarity with SWA yields results close to the worst. Moreover, CF

WBDTW even did not make it as a part of the best hybrid algorithm: The best combination turns out to be with CF AABovlGp03.

(9) *Simple ordered USS.* When generating 10 recommendations per query, there is no difference in precision-recall of Flat and the GP ordered weighting schemes with $r = 0.3$ and $r = 0.5$. Gp03 ($r = 0.3$) is slightly better when one takes only the top 1.

(10) *The simplest ordered USS.* Somewhat surprisingly, the simplest approach (OLQ) often outperforms the ones considering all queries: The best CF TkS method according to P_{AABovl}, R_{AABovl} and P_{AABcl} for SampledLog is CF AABclOLQ.

(11) *Benefit of hybridization.* As discussed, TnS and hybrid approaches can surpass the 'Best possible'.Some combinations do. For instance, CF AABovlGp03 with CF TnS and CB AABcl has a P_{WB} (0.198) on SampledLog. That is higher than what one can expect when retrieving recommendations only from the log (0.188). However, TnS alone did not win over 'the best possible'. Thus, one should use all proposed methods to achieve the best results in combination.

6.2.4.3 Runtime Characteristics

To quantify the runtime of the algorithms, we have measured average and maximum runtimes of giving recommendations for a user session. The preprocessing steps such as (1) computing WB representations, i.e., re-querying, and (2) getting AAB representations and computing proximity matrices, are excluded. These steps take place only once before running the recommendation engine. Table 6.4 reports the outcomes. As expected, the runtimes of the OLQ approaches are best. The unordered schemes and complex ordered ones (DTW, SWA) are noticeably slower than simple ordered methods. Indeed, their complexity of comparing two user sessions is $O(m \times n)$, where m and n are the lengths of corresponding user sessions. With WB QueRIE, the runtime is already high, even on the sampled data. For simple ordered USSs (Float and GP) the complexity is $O(min(m,n))$, whereas the complexity of OLQ is $O(1)$.

6.2.5 Discussion

The highest quality is achieved with hybridization. When using a hybrid combination, one should choose the one with the highest quality and acceptable response time. For instance, CF AABovlOLQ is the best CF TkS method according to most precision and recall values. In cases where it is not best, it is second. Taking its low runtime into account, it is a clear winner. CB AABcl has been the best CB method according to all precision and recall values. Thus, our best combination is CF AABovlOLQ + CF TnS + CB AABcl.

‡Flat, Gp05 and Gp03 always yield the same top 10 recommendations per query as Flat. This also is the case for Sampled log, with AABcl and WB query similarities.

Table 6.4: Average and maximum waiting time, in ms

Value / Method	t_{avg} **Full**	t_{max} **Full**	t_{avg} **Sampled**	t_{max} **Sampled**
CF AABovlGp03[§]	21	25,422	3.8	18,715.2
CF AABovlDTW	64.7	237,30	7.5	23,730.9
CF AABovlSWA	275.2	176,80	17.9	19,194
CF AABovlOLQ	1.3	**1.5**	**0.1**	**0.2**
CF AABovlUn	84.7	27,053	5.3	19,386.2
CF AABclGp03	372.5	124,590	144.2	99,437.1
CF AABclDTW	1,551	129,029	258	879,58.1
CF AABclSWA	1,424	85,741	402.7	95,854.6
CF AABclOLQ	1.5	**1.5**	0.2	**0.2**
CF AABclUn	1,665	143,22	228.3	100,082.5
CF WBGp03	8.7	13,295	4.3	15,826.7
CF WBDTW	19.4	14,155	3.4	10,250.6
CF WBSWA	81.3	14,010	9.9	10,839.8
CF WBOLQ	**0.7**	**1.5**	**0.1**	**0.2**
CF WBUn	14.9	10,385	3.3	11,842.7
CF TnS	890.3	12,444	82.7	1,309.7
CB AABovl	0.8	2	0.2	0.3
CB AABcl	0.9	2.2	0.3	0.5
CB WB	**0.7**	1.6	**0.1**	**0.2**
WB QueRIEWeig			14,081	805,106.8
WB QueRIEBin			14,081	805,106.8

6.3 Conclusions

We have developed approaches for data aware SQL query recommendation. Such approaches must be scalable and data-aware, i.e., a recommendation system should provide a query which can be issued directly. We have classified the existing approaches and have developed our owns by combining good design decisions with new ideas. We address the issue of quality metrics for query recommendations and have come up with metrics depending on the similarity function. We have tested our approaches against the SkyServer query log. Both performance and runtime experiments indicate that our methods outperform their competitors. To facilitate reproducibility, the data for experiments is available online.

[§]Runtimes of hybrid methods are the sums of runtimes of the pure methods. The runtimes of -Gp05 and -Gp03 are always very close to -Flat.

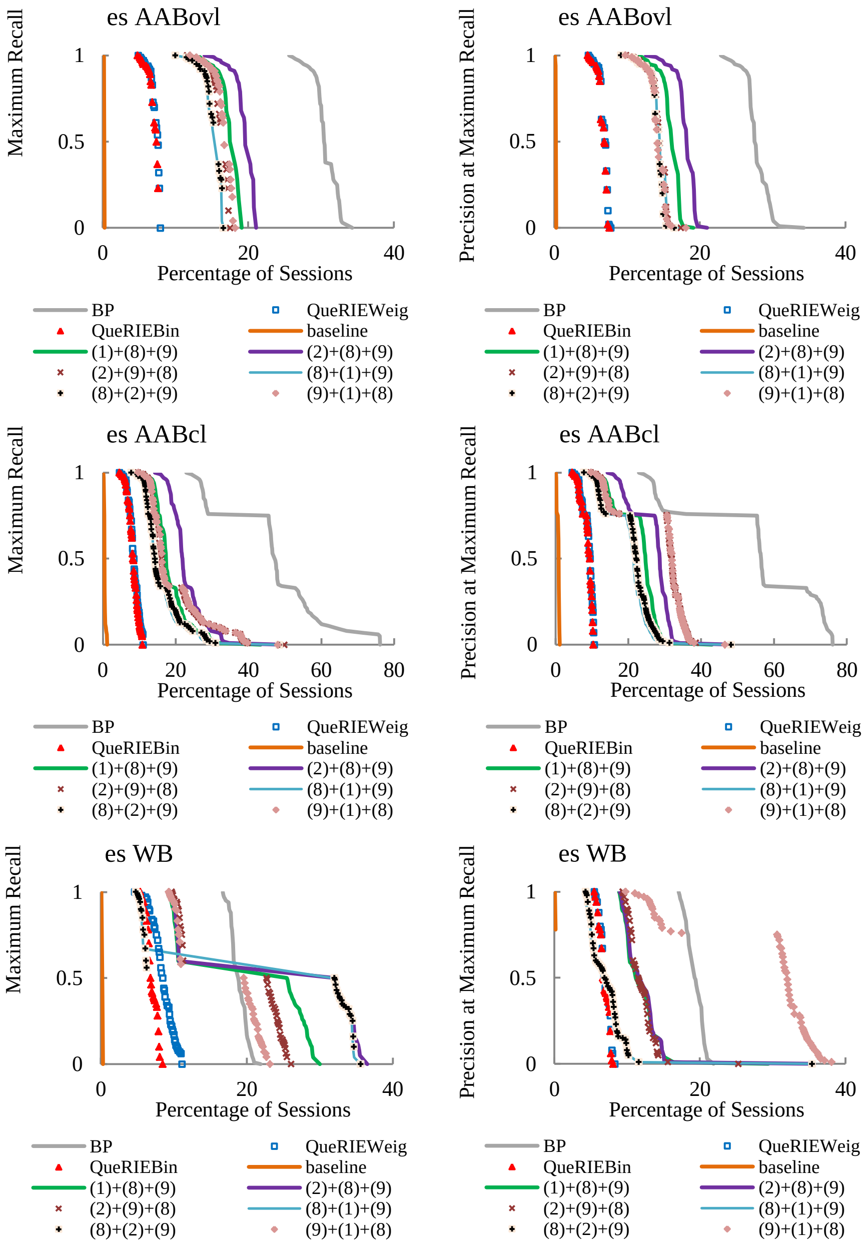

Figure 6.2: Inverse CDF of recall and precision at maximum recall, SampledLog

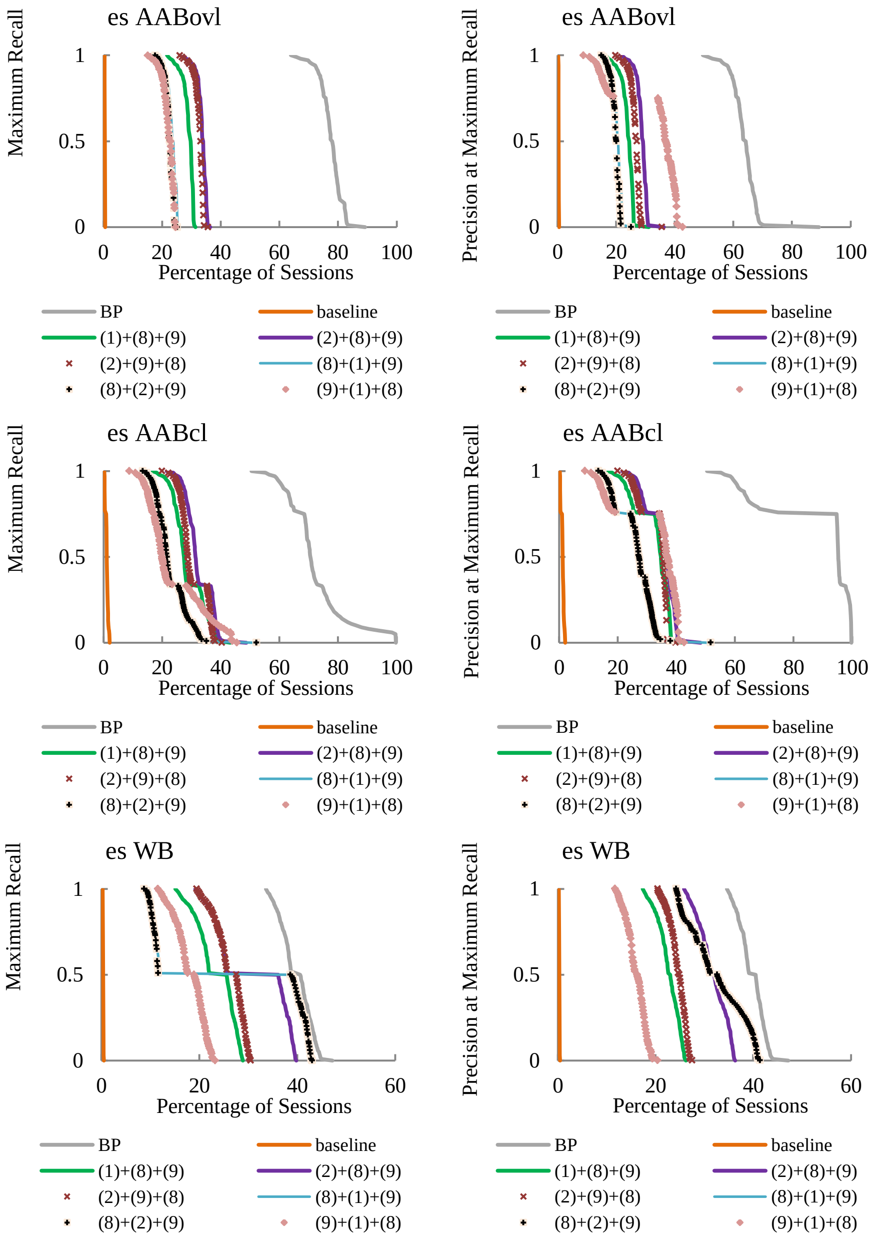

Figure 6.3: Inverse CDF of recall and precision at maximum recall, FullLog

7 Conclusion

SQL query logs analysis studies user interactions with databases. It applies to various scenarios: finding antipatterns in a log, clustering queries, SQL query recommendations.

First, we have performed a meta-analysis of the log and have identified patterns in it. Among them, we have distinguished antipatterns and sliding window search (SWS). This work was a necessary step before performing any other query log analysis.

Secondly, we have studied various SQL query representations and corresponding query similarity functions. When clustering with a particular similarity function, we have aimed to achieve excellent precision (match ground truth) and interpretability. We have motivated the necessity of proposing new data-aware similarity, which does not utilize the "heavy" witness-based (WB) approach. Our experiments show that our proposed AAB similarities outperform the competitors in both precision and interpretability.

Thirdly, we have proposed a new scalable and data-aware SQL query recommendation system (DASQR). We have highlighted the difference between standard and SQL query recommendations. Being aware of this difference, we have analyzed which design decisions are needed to create such a system. For each design decision, we have listed existing variants and proposed our own. The experiments indicate that our DASQR algorithms outperform the competitors in quality and runtime.

List of Algorithms

List of Figures

List of Tables

List of Theorems, Lemmas and Definitions

Bibliography

[AB] Natalia Arzamasova and Klemens Böhm. “Scalable and data-aware SQL query recommendations”. [see pages 97, 145]

[Agg+99] Charu C Aggarwal et al. “Horting hatches an egg: A new graph-theoretic approach to collaborative filtering”. In: *ACM SIGKDD*. 1999. [see page 29]

[Ali+14] Julien Aligon et al. “Similarity measures for OLAP sessions”. In: *KAIS* 39.2 (2014). [see pages 16, 34]

[Ali+15] Julien Aligon et al. “A collaborative filtering approach for recommending OLAP sessions”. In: *Decision Support Systems* 69 (2015). [see pages 5, 13, 34, 95, 107]

[AM+09] Ratish Agarwal, Dr Motwani, et al. “Survey of clustering algorithms for MANET”. In: *arXiv preprint arXiv:0912.2303* (2009). [see page 62]

[Arz+19] Natalia Arzamasova, Klemens Böhm, Bertrand Goldman, Christian Saaler, and Martin Schäler. “On the Usefulness of SQL-Query-Similarity Measures to Find User Interests”. In: *IEEE Transactions on Knowledge and Data Engineering* (2019). IEEE. [see pages 8, 61, 106, 145]

[ASB18] Natalia Arzamasova, Martin Schäler, and Klemens Böhm. “Cleaning antipatterns in an SQL query log”. In: *IEEE Transactions on Knowledge and Data Engineering* 30.3 (2018), pages 421–434. IEEE. [see pages 37, 145]

[AT05] Gediminas Adomavicius and Alexander Tuzhilin. “Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions”. In: *IEEE TKDE* 6 (2005). [see page 29]

[BG06] Stefan Brass and Christian Goldberg. “Semantic errors in SQL queries: A quite complete list”. In: *Journal of Systems and Software* 79.5 (2006), pages 630–644. Elsevier. [see page 17]

[BS97] Marko Balabanović and Yoav Shoham. “Fab: content-based, collaborative recommendation”. In: *Communications of the ACM* 40.3 (1997). [see page 29]

[Cao+08] Huanhuan Cao, Daxin Jiang, Jian Pei, Qi He, Zhen Liao, Enhong Chen, and Hang Li. “Context-aware query suggestion by mining click-through and session data”. In: *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 875–883. ACM. 2008. [see page 15]

[CEP09] Gloria Chatzopoulou, Magdalini Eirinaki, and Neoklis Polyzotis. “Query recommendations for interactive database exploration”. In: *SSDBM*. 2009. [see pages 6–8, 13, 16, 22, 28, 30, 31, 33, 34, 97, 109]

[Cha+15] Bikash Chandra, Bhupesh Chawda, Biplab Kar, KV Reddy, Shetal Shah, and S Sudarshan. “Data generation for testing and grading SQL queries”. In: *The VLDB Journal—The International Journal on Very Large Data Bases* 24.6 (2015), pages 731–755. Springer-Verlag New York, Inc.. [see page 78]

[Che+14] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. “Detecting performance anti-patterns for applications developed using object-relational mapping”. In: *Proceedings of the 36th International Conference on Software Engineering*, pages 1001–1012. ACM. 2014. [see page 17]

[Che+16] Tse-Hsun Chen, Weiyi Shang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. “Detecting problems in the database access code of large scale systems-an industrial experience report”. In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 71–80. IEEE. 2016. [see page 17]

[CL07] Zhiyuan Chen and Tao Li. “Addressing diverse user preferences in SQL-query-result navigation”. In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 641–652. ACM. 2007. [see page 16]

[Def77] Daniel Defays. “An efficient algorithm for a complete link method”. In: *The Computer Journal* 20.4 (1977), pages 364–366. Oxford University Press. [see page 77]

[Dud+03] Bill Dudney, Stephen Asbury, Joseph K Krozak, and Kevin Wittkopf. *J2EE antipatterns*. John Wiley & Sons, 2003. [see pages 1, 38]

[Ees15] Erki Eessaar. “On query-based search of possible design flaws of SQL databases”. In: *Innovations and Advances in Computing, Informatics, Systems Sciences, Networking and Engineering*. Springer, 2015, pages 53–60. [see page 17]

[Eir+14] Magdalini Eirinaki et al. “Querie: Collaborative database exploration”. In: *IEEE TKDE* 26.7 (2014). [see pages 3, 13, 21, 22, 28, 31, 33, 34, 40, 95, 97, 101, 107, 137, 138]

[Est+96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. “A density-based algorithm for discovering clusters in large spatial databases with noise.” In: *Kdd*. Volume 96. 34, pages 226–231. 1996. [see page 77]

[Fow97] Martin Fowler. “Refactoring: Improving the design of existing code”. In: *11th European Conference. Jyväskylä, Finland*. 1997. [see pages 1, 17]

[GS09] Asela Gunawardana and Guy Shani. “A survey of accuracy evaluation metrics of recommendation tasks”. In: *MLR* 10.Dec (2009). [see page 107]

[HBV01] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. "On clustering validation techniques". In: *Journal of intelligent information systems* 17.2-3 (2001), pages 107–145. Springer. [see page 3]

[HBV02] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. "Cluster validity methods: part I". In: *ACM Sigmod Record* 31.2 (2002), pages 40–45. ACM. [see page 81]

[II08] Aminul Islam and Diana Inkpen. "Semantic text similarity using corpus-based word similarity and string similarity". In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 2.2 (2008), page 10. ACM. [see page 20]

[KA10] Bill Karwin and SQL Antipatterns. "Avoiding the Pitfalls of Database Programming". In: *The Pragmatic Bookshelf* (2010), pages 15–155. [see page 48]

[KBGM09] Georgia Koutrika, Benjamin Bercovitz, and Hector Garcia-Molina. "FlexRecs: Expressing and Combining Flexible Recommendations". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09, pages 745–758. ISBN: 978-1-60558-551-2. Providence, Rhode Island, USA: ACM, 2009. [see pages 32, 34, 107]

[Kho+10] Nodira Khoussainova et al. "SnipSuggest: Context-aware autocompletion for SQL". In: *VLDBJ* 4.1 (2010). [see pages 3, 5, 8, 16, 21, 28, 33, 34, 40, 102, 107, 138]

[Kul+18] Gokhan Kul, Duc Thanh Anh Luong, Ting Xie, Varun Chandola, Oliver Kennedy, and Shambhu Upadhyaya. "Similarity metrics for sql query clustering". In: *IEEE Transactions on Knowledge and Data Engineering* 30.12 (2018), pages 2408–2420. IEEE. [see page 78]

[Mat+10] Sunu Mathew, Michalis Petropoulos, Hung Q Ngo, and Shambhu Upadhyaya. "A data-centric approach to insider attack detection in database systems". In: *International Workshop on Recent Advances in Intrusion Detection*, pages 382–401. Springer. 2010. [see page 22]

[Ngu+15] Hoang Vu Nguyen et al. "Identifying User Interests within the Data Space-a Case Study with SkyServer." In: *EDBT*. 2015. [see pages 3, 4, 7, 16, 23, 24, 28, 57, 58, 66]

[PJ09] Hae-Sang Park and Chi-Hyuck Jun. "A simple and fast algorithm for K-medoids clustering". In: *Expert systems with applications* 36.2 (2009), pages 3336–3341. Elsevier. [see page 77]

[Rad+14a] M Jordan Raddick, Ani R Thakar, Alexander S Szalay, and Rafael DC Santos. "Ten years of SkyServer I: Tracking web and SQL e-science usage". In: *Computing in Science & Engineering* 16.4 (2014), pages 22–31. IEEE. [see page 16]

[Rad+14b] M Jordan Raddick, Ani R Thakar, Alexander S Szalay, and Rafael DC Santos. “Ten years of SkyServer II: How astronomers and the public have embraced e-science”. In: *Computing in Science & Engineering* 16.4 (2014), pages 32–40. IEEE. [see page 16]

[Rou87] Peter J Rousseeuw. “Silhouettes: a graphical aid to the interpretation and validation of cluster analysis”. In: *Journal of computational and applied mathematics* 20 (1987), pages 53–65. Elsevier. [see page 81]

[RRS11a] Francesco Ricci, Lior Rokach, and Bracha Shapira. “Introduction to recommender systems handbook”. In: *Recommender systems handbook*. 2011. [see pages 5, 29]

[RRS11b] Francesco Ricci, Lior Rokach, and Bracha Shapira. “Introduction to recommender systems handbook”. In: *Recommender systems handbook*. Springer, 2011, pages 1–35. [see page 107]

[Sak+90] Hiroaki Sakoe et al. “Dynamic programming algorithm optimization for spoken word recognition”. In: *Readings in speech recognition* 159 (1990). [see pages 8, 100]

[SBM96] Michael Stal, Frank Buschmann, and Regine Meunier. *Pattern-oriented Software Architecture—A System of Patterns*. 1996. [see page 19]

[SF02] Mikael Sollenborn and Peter Funk. “Category-based filtering and user stereotype cases to reduce the latency problem in recommender systems”. In: *European Conference on Case-Based Reasoning*. 2002. [see page 98]

[Sil+09] Fabrizio Silvestri et al. “Mining query logs: Turning search usage data into knowledge”. In: *Foundations and Trends® in Information Retrieval* 4.1–2 (2009), pages 1–174. Now Publishers, Inc.. [see page 15]

[Sin+07] Vik Singh, Jim Gray, Ani Thakar, Alexander S Szalay, Jordan Raddick, Bill Boroski, Svetlana Lebedeva, and Brian Yanny. “Skyserver traffic report-the first five years”. In: *arXiv preprint cs/0701173* (2007). [see pages 16, 30, 53]

[SW00] Connie U Smith and Lloyd G Williams. “Software performance antipatterns.” In: *Workshop on Software and Performance*. Volume 17, pages 127–136. Ottawa, Canada. 2000. [see pages 17, 19]

[SW+81] Temple F Smith, Michael S Waterman, et al. “Identification of common molecular subsequences”. In: *Journal of molecular biology* 147.1 (1981). [see pages 8, 34, 100]

[TK11] Catia Trubiani and Anne Koziolek. “Detection and solution of software performance antipatterns in palladio architectural models”. In: *ACM SIGSOFT Software Engineering Notes* 36.5 (2011), pages 36–36. ACM. [see pages 18, 19]

[Wer+14] Alexander Wert, Marius Oehler, Christoph Heger, and Roozbeh Farahbod. “Automatic detection of performance anti-patterns in inter-component communications”. In: *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures*, pages 3–12. ACM. 2014. [see pages 17–19]

[WMW17] Xiaolan Wang, Alexandra Meliou, and Eugene Wu. “QFix: Diagnosing errors through query histories”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1369–1384. ACM. 2017. [see pages 15, 18]

[WW07] Silke Wagner and Dorothea Wagner. *Comparing clusterings: an overview.* Universität Karlsruhe, Fakultät für Informatik Karlsruhe, 2007. [see page 81]

[YPS09] Xiaoyan Yang, Cecilia M Procopiuc, and Divesh Srivastava. “Recommending join queries via query log analysis”. In: *IEEE ICDE*. 2009. [see pages 16, 32–34, 107]

[ZMJM14] Mohammed J Zaki, Wagner Meira Jr, and Wagner Meira. *Data mining and analysis: fundamental concepts and algorithms.* Cambridge University Press, 2014. [see page 81]

List of Notations

Notation	Description	The first mention
DB	A relational database	Definition 2.1, page 11
S	A database schema	Definition 2.2, page 11
T	A database state	Definition 2.3, page 11
q	An SQL query	Definition 2.5, page 11
a	An attribute a	Section 2.1
$dom(a)$	A domain of an attribute a	Definition 2.6, page 11
U	The universal relation	Definition 2.7, page 11
$U(q)$	The universal relation	Section 2.2, Definition 2.8, page 11
$P(q)$	Filtering conditions of a query q	Section 2.2, Definition 2.9, page 11
$Ints(q)$	The interest of a query q	Section 2.2, Definition 2.10, page 12
$operators(q)$	The operators of a query q	Section 2.2, Definition 2.12, page 12
$operators(q)[ints]$ or $operators(q)[i]$	Operators of the interest $ints$	Section 2.2, Definition 2.14, page 12
$P(q)[ints]$ or $P(q)[i]$	Filtering conditions of the interest	Section 2.2, Definition 2.16, page 12
us_i	A user session	2.18
US	A query log	Section 2.3, Definition 2.19, page 13
QRS	A query representation scheme	Section 2.4, Definition 2.20, page 13
$QR(q)$	A query representation of a query	Section 2.4, Definition 2.21, page 13
us_0	The *current* user session	Section 2.5, Definition 2.23, page 13
q^0_{n+1}	An unseen query	Section 2.5, Definition 2.24, page 13
nr	A number of recommendations an SQL recommendation system should provide	Section 2.5
ϕ_j	A feature in DB allowed by schema S	Section 3.3.2
ϕ^i	A FB-query from [Eir+14]: a vector of features $(\phi^i_1, \ldots, \phi^i_k)$ whose cell ϕ^i_j contains a weight if the feature ϕ_j appears in q_i	Section 3.3.2
τ_j	A tuple in DB of state T	Section 3.3.4
τ^i	A WB-query: a vector of tuples $(\tau^i_1, \ldots, \tau^i_k)$ whose cell τ^i_j represents the importance of thetuple τ_j as a witness for q_i	Section 3.3.2
$(U,T)P$	The result set of query q	Section 3.3.4
$(U\backslash t\}, T)P$	t is removed from U	Section 3.3.4
$\sigma_{P(q)}(U(q))$	The access area of a query q	Section 3.3.4, Definition 3.6, page 23

$d_{tables}(q_1.FROM, q_2.FROM)$	The distance of tables in q_1 and q_2	Section 3.3.4, Definition 3.7, page 24
$d_{conj}(b_1; b_2)$	distance of conjunctions of two queries q_1 and q_2 with predicates b_1 and b_2 (in CNF)	Section 3.3.4, Definition 3.8, page 24
$d_{disj}(o_1; o_2)$	distance of disjunctions, each $o_i \in b_i$	Section 3.3.4, Definition 3.9, page 24
$d_{pred}(p_1, p_2)$	distance of two atomic predicates p_1 and p_2, $p_1 \in o_1$	Section 3.3.4
$S(q_1, q_2)$	Query similarity	Section 3.3
$USS(us_1, us_2)$	User session similarity	Section 3.3
us_0	A current user session (one, we need to have a prediction for)	Definition 2.23, Section 2.5
S_i	A WB or FB summary of a user session us_i	Section 3.4.3.1 or 3.4.3.5
S_0^{pred}	The predicted summary (WB or FB) of us_0	Section 3.4.3.1 or 3.4.3.5
$USS_{WB}(S_i, S_i)$	WB user session similarity (Jaccard or cosine similarity)	Section 3.4.3.1
$s\phi_i$	FB-query according to [Kho+10]: a sequence of features	Section 3.4.3.5

$sim(\rho, \psi)$	Similarity of two features ρ and ψ	Section 3.4.3.5
F_ρ	A vector $(w_1^\rho, \ldots, w_n^\rho)$ of the presence of feature ρ in all user sessions $\{S_1, \ldots, S_n\}$	Section 3.4.3.5
t	A template of a query q: a FB query representation [Eir+14] of q	Section 4.1.1.2, Definition 4.5, page 40
p	A pattern as a sequence of query templates $(t_1, \ldots, t_n)$	Section 4.1.1.2, Definition 4.3, page 39
t^{SELECT}	All features of a query q, which start from f^{SELECT}	Section 4.1.1.2, Definition 4.6, page 40
t^{FROM}	All features of a query q, which start from f^{FROM}	Section 4.1.1.2, Definition 4.7, page 40
t^{WHERE}	All features of a query q, which start from f^{WHERE}	Section 4.1.1.2, Definition 4.8, page 40
$t^{GROUPBY}$	All features of a query q, which start from $f^{GROUPBY}$	Section 4.1.1.2, Definition 4.9, page 40
$u = user(q)$	A user, who has run q	Section 4.1.1.3, Definition 4.10, page 40
$time(q)$	A moment in time, when q was requested.	Section 4.1.1.3, Definition 4.11, page 40
$(q_1, \ldots, q_n)$	An instance of a pattern $p = (t_1, \ldots, t_n)$	Section 4.1.1.3, Definition 4.12, page 40
$frequency$	The *frequency* of a pattern	Section 4.1.1.3, Definition 4.13, page 41
$userPopularity$	The *userPopularity* of a pattern	Section 4.1.1.3, Definition 4.14, page 41
Stifle	A *Stifle* antipattern	Section 4.1.2.1, Definition 4.15, page 42

DW-Stifle	A *DW-Stifle* antipattern	Section 4.1.2.1, Definition 4.17, page 42
DS-Stifle	A *DS-Stifle* antipattern	Section 4.1.2.1, Definition 4.20, page 43
DF-Stifle	A *DF-Stifle* antipattern	Section 4.1.2.1, Definition 4.23, page 43
CTH candidate	A CTH antipattern candidate	Section 4.1.2.2, Definition 4.25, page 44
$D(q_1, q_2)$	A distance of q_1 and q_2 $D(q_1, q_2) \in [0; 1]$; $D(q_1, q_2) = 1 - S(q_1, q_2)$	Section 5.1.1
$D(q_1, q_2).a$	The distance of two queries q_1, q_2 with respect of Attribute a	Section 5.1.3, Definition 5.3, page 63
OA	An ordinal attribute	Section 5.1.3, Definition 5.4, page 63
NA	A nominal attribute	Section 5.1.3, Definition 5.5, page 63
$comInts(q_1, q_2)$	The common interest of two queries q_1 and q_2	Section 5.1.3, Definition 5.6, page 63
$exclInts(q_1, q_2)$	The exclusive interest of two queries q_1 and q_2	Section 5.1.3, Definition 5.7, page 63
A_i^{OA}	A set of intervals of a query q_i with an ordinal Attribute a	Section 5.1.3, Definition 5.8, page 63
$width(a_{i.k})$	The width of interval $a_{i.k}^{OA}$	Section 5.1.3, Definition 5.11, page 64
A_i^{NA}	The set of valid values with a nominal Attribute a	Section 5.1.3, Definition 5.12, page 64
$S_{cl}(q_1, q_2).a$	The similarity of two queries with the same attribute	Section 5.1.5.1, Definition 5.14, page 66
$comWidth$ $(a_{1.1}^{OA}, a_{2.1}^{OA})$	The with of an overlap interval two queries q_1 and q_2 with the only occurrence of an Attribute a have	Section 5.1.5.2, Formula 5.5, page 68
$allWidth$ $(a_{1.1}^{OA}, a_{2.1}^{OA})$	The width of interval two queries q_1 and q_2 with the only occurrence of an Attribute a have	Section 5.1.5.3, Formula 5.6, page 68
$overallComWidth$ (a_1, a_2)	The overall width of overlap interval two queries q_1 and q_2 with an Attribute a have	Section 5.1.6.1, Formula 5.8, page 70
$overallAllWidth$ (a_1, a_2)	The overall width of interval two queries q_1 and q_2 an Attribute a have	Section 5.1.6.1, Formula 5.9, page 70
$S_{attrCom}(q_1, q_2)$	Similarity for common interests of queries q_1 and q_2	Section 5.1.7.1, Formula 5.11, page 72
$S_{attrExcl}(q_1, q_2)$	Similarity for exclusive interests of queries q_1 and q_2	Section 5.1.7.2, Formula 5.12, page 73
$S_{attrFull}(q_1, q_2)$	Overall attribute similarity	Section 5.1.7.3, Formula 5.13, page 74
$reductCoeff_{table}$	Reduction table coefficient	Section 5.1.8, Formula 5.16, page 75
TkS	CF thick query similarity, no fitting (TkS)	Section 6.1.2

TnS	CF thin query similarity, thick sitting (TnS)	Section 6.1.3
us_1	The user session we compare us_0 to	Section 6.1.2.2
$\|us_0\| = n+1$	The length of us_0	Section 6.1.2.2
$\|us_1\| = m+1$	The length of us_1	Section 6.1.2.2
q^0_{n+1}	An unseen query	Section 6.1.2.2
q^1_{m+1}	A possible prediction for q^0_{n+1}	Section 6.1.2.2
$USS_{flat}(us_0, us_1)$	Flat user session similarity	Section 6.1.2.2, Formula 6.1, page 100
$USS_{GP}(us_0, us_1)$	Geometric progression user session similarity	Section 6.1.2.2, Formula 6.2, page 100
$USS_{OLQ}(us_0, us_1)$	"Only last query" user session similarity	Section 6.1.2.2, Formula 6.3, page 100
SWA	The Smith-Waterman algorithm	Section 6.1.2.2
DTW	Dynamic time warping algorithm	Section 6.1.2.2
$USS_{unord}(us_0, us_1)$	Unordered user session similarity	Section 6.1.2.2, Formula 6.4, page 101
$FB\ QueRIE\ TR$	FB QueRIE recommendation	Section 6.1.3.1
$GBTR$	Graph-based template recommendation	Section 6.1.3.1
GoT	Graph of templates	Section 6.1.3.1, Formula 6.2, page 102
$weigth(t_i, t_j)$	The weight of an arrow indicates the probability of having t_j immediately after t_i in a query log US	Section 6.1.3.1
$tswp$	A set of template sequences (patterns) with corresponding probabilities	Section 6.1.3.1
qr_i	A recommended query	Section 6.1.3.2
$Fitting((q_1, q_2), q^0_n)$	The fitting procedure	Section 6.1.3.2, Algorithm 6.4, page 104
a'	A related attribute	Section 6.1.3.2, Definition 6.3, page 103
ar_{shift}	The shift of an interval in qr_i for an attribute a	Section 6.1.3.2
$width(ar_i)$	The width of an interval in qr_i for an attribute a	Section 6.1.3.2
P	Precision	Section 6.2.3
R	Recall	Section 6.2.3
P_{WB}	WB precision	Section 6.2.3.1, Formula 6.10, page 109
R_{WB}	WB recall	Section 6.2.3.1, Formula 6.10, page 109
$P_{AABovl}.a^{OA}$	AABovl precision for an OA a	Section 6.2.3.1, Formula 6.12, page 109
$R_{AABovl}.a^{OA}$	AABovl recall for an OA a	Section 6.2.3.1, Formula 6.13, page 109
$P_{AABovl}.a^{NA}$	AABovl precision for a NA a	Section 6.2.3.1, Formula 6.14, page 109
$R_{AABovl}.a^{NA}$	AABovl recall for a NA a	Section 6.2.3.1, Formula 6.15, page 109
$P_{AABcl}.a^{OA}$	AABcl precision for an OA a	Section 6.2.3.1, Formula 6.16, page 110
$R_{AABcl}.a^{OA}$	AABcl recall for a OA a	Section 6.2.3.1, Formula 6.17, page 110

$P_{attrFull}(q_U, q_R)$	AAB precision for all attributes in the filtering conditions of q_U and q_R	Section 6.2.3.1, Formula 6.18, page 110
$P_{attrCom}(q_U, q_R)$	AAB precision for common attributes in the filtering conditions of q_U and q_R	Section 6.2.3.1, Formula 6.17, page 110
$P_{attrExcl}(q_U, q_R)$	AAB precision for exclusive attributes in the filtering conditions of q_U and q_R	Section 6.2.3.1, Formula 6.20, page 110
$R_{attrFull}(q_U, q_R)$	AAB recall for all attributes in the filtering conditions of q_U and q_R	Section 6.2.3.1, Formula 6.22, page 110
$R_{attrCom}(q_U, q_R)$	AAB recall for common attributes in the filtering conditions of q_U and q_R	Section 6.2.3.1, Formula 6.23, page 110
$R_{attrExcl}(q_U, q_R)$	AAB recall for exclusive attributes in the filtering conditions of q_U and q_R	Section 6.2.3.1, Formula 6.24, page 110
$P(q_U, q_R)$	AAB precision	Section 6.2.3.1, Formula 6.21, page 110
$R(q_U, q_R)$	AAB recall	Section 6.2.3.1, Formula 6.25, page 110
ρ_s	The Spearman correlation	Section 6.2.4.2

Curriculum Vitae

Personal Data

Birth	on 18th of January, 1989 in Cheboksary, Russia
Nationality	Russian

University Education

2013-2019	PhD student in computer science at the Karlsruhe Institute of Technology in the research group for Program Structures and Data Organisation of Prof. Dr. Klemens Böhm
2005-2010	Master diploma of computer science at Chuvash State University (Cheboksary, Russia).

List of Publications

In Journal Proceedings

Natalia Arzamasova, Martin Schäler, and Klemens Böhm. “Cleaning antipatterns in an SQL query log”. In: *IEEE Transactions on Knowledge and Data Engineering* 30.3 (2018), pages 421–434. IEEE

Natalia Arzamasova, Klemens Böhm, Bertrand Goldman, Christian Saaler, and Martin Schäler. “On the Usefulness of SQL-Query-Similarity Measures to Find User Interests”. In: *IEEE Transactions on Knowledge and Data Engineering* (2019). IEEE

Natalia Arzamasova and Klemens Böhm. “Scalable and data-aware SQL query recommendations”