Similarity Metrics for SQL Query Clustering

Gokhan Kul, *Graduate Student Member, IEEE*, Duc Thanh Anh Luong, Ting Xie, Varun Chandola, Oliver Kennedy, *Member, IEEE*, and Shambhu Upadhyaya, *Senior Member, IEEE*

Abstract—Database access logs are the starting point for many forms of database administration, from database performance tuning, to security auditing, to benchmark design, and many more. Unfortunately, query logs are also large and unwieldy, and it can be difficult for an analyst to extract broad patterns from the set of queries found therein. Clustering is a natural first step towards understanding the massive query logs. However, many clustering methods rely on the notion of pairwise similarity, which is challenging to compute for SQL queries, especially when the underlying data and database schema is unavailable. We investigate the problem of computing similarity between queries, relying only on the query structure. We conduct a rigorous evaluation of three query similarity heuristics proposed in the literature applied to query clustering on multiple query log datasets, representing different types of query workloads. To improve the accuracy of the three heuristics, we propose a generic feature engineering strategy, using classical query rewrites to standardize query structure. The proposed strategy results in a significant improvement in the performance of all three similarity heuristics.

Index Terms—Clustering; Query Logs; Similarity Metric; Summarization

1 Introduction

Database access logs are used in a wide variety of settings, including evaluating database performance tuning [1], benchmark development [2], database auditing [3], and compliance validation [4]. Also, many user-centric systems utilize query logs to help users by providing recommendations and personalizing the user experience [5], [6], [7], [8], [9], [10]. As the basic unit of interaction between a database and its users, the sequence of SQL queries that a user issues effectively models the user's behavior. Queries that are similar in structure imply that they might be issued to perform similar duties. Examining a history of the queries serviced by a database can help database administrators with tuning, or help security analysts to assess the possibility and/or extent of a security breach. However, logs from enterprise database systems are far too large to examine manually. As one example, a recent study of queries at a major US bank for a period of 19 hours found nearly 17 million SQL queries and over 60 million stored procedure execution events [3]. Even excluding stored procedures, it is unrealistic to expect any human to manually inspect all 17 million queries per day.

Let us consider an analyst (call her Jane) faced with the task of analyzing such a query log. Jane might first attempt to identify some interesting query fragments and their aggregate properties. For example, she might count how many times each table is accessed or the frequency with which different classes of join predicates occur. Unfortunately, such fine-grained properties lack the context to clearly communicate how the data is being used, combined, and/or manipulated. To see the complete context, Jane must look at entire queries. Naively, she might look at all *distinct query strings* in the log. Even comparatively small production databases typically log hundreds or thousands of distinct query strings, making direct inspection impractical. Furthermore, it is unclear that distinct query strings are the right level of granularity in the first place. Consider the following example queries:

```
1) SELECT name FROM user
   WHERE rank IN ('adm','sup')
2) SELECT SUM(balance) FROM accounts
3) SELECT name FROM user WHERE rank = 'adm'
       UNION SELECT name FROM user
             WHERE rank = 'sup'
4) SELECT SUM(accounts.balance) FROM accounts
       NATURAL JOIN user WHERE user.rank = 'adm'
```

Queries 1 and 2 are clearly distinct: Their structures differ, they reference different datasets, and perform different computations. The remaining queries however are less so. Query 3 is logically equivalent to Query 1: Both compute identical results. Conversely, although Query 4 is distinct from Queries 1 and 2, it is conceptually similar to both and shares many structural features with each.

The exact definition of similarity may depend on Jane's exact task, the content of the log, the database schema, database records, and numerous other details, some of which may not be available to Jane immediately when she first begins analyzing the log. It is also likely that some of this information, like the precise contents of the database or even the database schema may not even be available to Jane for reasons of privacy or security. As a result, this type of log analysis can quickly become a tedious, time-consuming process [11]. An earlier work of Aligon et al. [12] attempted to address this problem for OLAP operations by performing query log analysis and exploration. Within the scope of this article, we focus on analysis of SQL queries instead of OLAP queries. In particular, we lay the groundwork for a more automated approach to SQL query log exploration based on hierarchical clustering. Given a hierarchical clustering of the SQL query log, Jane can manually adjust how aggressively

- *All the authors are with the Department of Computer Science and Engineering, University at Buffalo, Buffalo, NY, 14260.*
- *The first three authors contributed equally and can be considered a joint first author.*

the log is summarized. She can select an appropriate level of granularity without a *priori* needing to specify exactly what constitutes a similar query.

The primary focus of this article is to study the suitability of three existing query distance metrics [13], [14], [15] to be used with hierarchical algorithms for clustering query logs. All of these metrics operate on the query structure and do not rely on the availability of underlying data or schema, thus making them applicable in a wide variety of practical settings. We evaluate the three metrics on two types of data: Human-authored and Machine-generated. Thus, using an appropriate similarity metric, one can cluster the queries to obtain a meaningful clustering of the query log.

For our evaluation, we use three evaluation data sets: i) a large set of student authored queries released by IIT Bombay [16], ii) a smaller set of student queries gathered at the University at Buffalo, and released as part of this publication, and iii) SQL logs that capture all activities on 11 Android phones for a period of one month [2]. Student-written queries are appealing, as queries are already labeled by their ground-truth clusterings — For each question, the student is attempting to accomplish one specific stated task. Conversely, machine-generated queries on smartphones present a conceptually easier challenge, as they produce more rigid, structured queries. The three similarity metrics are evaluated on these data sets using three standard clustering evaluation statistics: Silhouette Coefficient, Beta CV, and Dunn Index [17].

None of the similarity metrics perform as well as desired, so we propose and evaluate a pre-processing step to create more regular, uniform query representations by leveraging query equivalence rules and data partitioning operations. These rules are commonly utilized by database management systems when parsing and evaluating SQL queries. This process significantly improves the quality of all three distance metrics. We also investigate and identify sources of errors in the clustering process. Experimental results show that our *regularization* pre-processing technique consistently improves clustering for different query comparison schemes from the literature.

Concretely, the specific contributions of this article are: (1) A survey of existing SQL query similarity metrics, (2) An evaluation of these metrics on multiple query logs, and (3) Applying query standardization techniques to improve query clustering accuracy.

This article is organized as follows. We start by performing a literature survey on log clustering and SQL query similarity in Section 2. We describe a feature engineering technique called regularization in Section 3. In Section 4, we explain our query workloads and propose a strategy for evaluating the quality of query similarity metrics. The evaluation is presented in Section 5. We discuss our experiment results, findings and ideas to further build upon the surveyed techniques in Section 6, and in Section 7, we explain how this work can be beneficial by giving real life examples. Finally, we conclude by identifying the steps needed to deploy query log clustering into practice using the techniques evaluated in this article in Section 8.

2 Background

Analyzing query logs mostly relies on the structure of queries [18], although their motivations are different; some methods prefer using the log as a resource to collect information to build user profiles, and the others utilize structural similarity to perform tasks like query recommendation [6], [7], [10], performance optimization [13], session identification [14] and workload analysis [15]. A summary of these methods is given in Table 1.

There are also other possible approaches; like data-centric query comparison [19], and utilizing the *access areas* of user queries by inspecting the data partition the query is interested in [20] from the `WHERE` condition. However, these approaches are out of our scope since we are interested in comparing and improving methods based on structural similarity; we assume that we do not have access to the data or the statistical information about the database.

Agrawal *et al.* [21] aim to rank the tuples returned by the SQL query based on the context. They create a ruleset for contexts and evaluate the result of queries that belongs to the context according to the ruleset. They capture context and query as feature vectors and capture similarity through cosine distance between the vectors.

Chatzopoulou *et al.* [10] aim to assist non-expert users of scientific databases by tracking their querying behavior and generating personalized query recommendations. They deconstruct an SQL query into a bag of *fragments*. Each distinct fragment is a feature, with a weight assigned to it indicating its importance. Each feature has two types of importance: (1) within the query and (2) for the overall workload. Similarity is defined upon common vector-based measures such as cosine similarity. A summarization/user profile for this approach is just a sum over all single query feature vectors that belong to their workload.

Yang *et al.* [7], on the other hand, build a graph following the query log by connecting associations of table attributes from the input and output of queries which are then used to compute the likelihood of an attribute appearing in a query with a similarity function like Jaccard coefficient. Their aim is again to assist users in writing SQL queries by analyzing query logs. Giacometti *et al.* [6], similarly, aim to make recommendations on the discoveries made in the previous sessions for users to spend less time on investigating similar information. They introduce *difference pairs* in order to measure the relevance of the previous discoveries. Difference pairs are essentially the result columns that is not included in the other return results; hence the method depends on having access to the data. Stefanidis *et al.* [8] takes a different approach, and instead of recommending candidate queries, they recommend tuples that may be of interest to the user. By doing so, the users may decide to change the selection criteria of their queries in order to include these results.

Sapia [5] creates a model that learns query templates to prefetch data in OLAP systems based on the user's past activity. SnipSuggest [9], on the other hand, is a context-aware SQL-autocomplete system that helps database users to write SQL queries by suggesting SQL snippets. In particular, it assigns a probability score to each subtree of a query based on the subtree's frequency in a query log. These probabilities

Paper title	Motivation	Features	Feature Structure	Distance Function	Similarity Ratio
Agrawal *et al.* (2006) [21]	Q. reply importance	Schema, rules	Vector	Cosine similarity	No
Giacometti *et al.* (2009) [6]	Q. recommendation	Difference pairs	Set	Difference query	No
Yang *et al.* (2009) [7]	Q. recommendation	Selection/join, projection	Graph	Jaccard coefficient on the graph edges	No
Stefanidis *et al.* (2009) [8]	Data Recommendation	Inner product of two queries	Vector	-	No
Khoussainova *et al.* (2010) [9]	Q. recommendation	Popularity of each query object	Graph	-	No
Chatzopoulou *et al.* (2011) [10]	Q. recommendation	Syntactic element frequency	Vector	Jaccard coefficient and cosine similarity	No
Aouiche *et al.* (2006) [13]	View selection	Selection/join, group-by	Vector	Hamming distance	Yes
Aligon *et al.* (2014) [14]	Session similarity	Selection/join, projection, group-by	3 Sets	Jaccard coefficient	Yes
Makiyama *et al.* (2016) [15]	Workload analysis	Term frequency of projection, selection/join, from, group-by and order-by	Vector	Cosine similarity	Yes

TABLE 1: SQL query similarity literature review

are used to discover the most likely subtree that a user is attempting to construct, at interactive speeds.

Although these methods [6], [7], [8], [9], [10], [21] utilize query similarity one way or other to achieve their purpose, they don't directly offer a way to compare query similarity. We aim to summarize the log and the most practical way to describe a query log is to group similar queries together so that we can provide summaries of these groups to the users. For this purpose, we need to be able to measure pairwise similarity between each query, hence we need a metric that can do so. As shown in Table 1, this condition is only satisfied by [13], [14], [15].

Aouiche *et al.* [13] is the first work we encountered that proposes a pairwise similarity metric between two SQL queries although it is not the aim of their work. They aim to optimize view selection in warehouses by the queries posed to the system. They consider the *selection*, *joins* and *group-by* items in the query to create vectors and use Hamming Distance to measure how similar two queries are. While creating the vector, it doesn't matter if an item appears more than once or where the item is. They cluster similar queries that creates a workload on the system and base their view creation strategy in the system on the clustering result.

Aligon *et al.* [14] study various approaches to defining a similarity function to compare OLAP sessions. They focus on comparing session similarity while also performing a survey on query similarity metrics. They identify *selection* and *join* items as the most relevant components in a query followed by the *group by* set. Inspired by the findings, they propose their own query similarity metric which considers *projection*, *group-by*, *selection-join* items for queries issued on OLAP datacubes. OLAP datacubes are multidimensional models, and they have hierarchy levels for the same attributes. Aligon *et al.* [14] measure the distance between the attributes on different hierarchy levels, and compute the set similarity for *projection*, *group-by*, and *selection-join* sets individually when comparing two queries. In our experiments, since we do not consider the hierarchy levels in an OLAP system but focus on databases, we consider all queries are on the same level in the schema to adjust the formulas presented in the paper. Namely, we compute set similarity of *projection*, *group-by*, *selection-join* sets of two queries with Jaccard coefficient. Also, Aligon *et al.* [14] provide the flexibility to adjust weights of the three feature sets based on the domain needs. We explore how the clustering quality is affected with various weightings in Appendix A.

Paper title	Extracted Feature Vector
Aouiche *et al.* (2006) [13]	{‘u.id’, ‘a.userid’, ‘a.balance’, ‘u.yearenrolled’}

Aligon *et al.* (2014) [14]	{‘u.username‘, ‘u.yearenrolled’}
	{‘u.id’, ‘a.userid’, ‘a.balance’}
	{‘u.yearenrolled’}

Makiyama *et al.* (2016) [15]	{‘SELECT_u.username’ →1, ‘SELECT_u.yearenrolled’ →1, ‘FROM_user →1’, ‘FROM_accounts’ →1, ‘WHERE_u.id’ →2, ‘WHERE_a.userid’ →1, ‘WHERE_a.balance’ →1, ‘GROUPBY_u.yearenrolled’ →1, ‘ORDERBY_u.yearenrolled’ →1}

TABLE 2: Representation of three similarity metrics

Makiyama *et al.* [15] approach query log analysis with the goal of analyzing a system's workload, and they provide a set of experiments on Sloan Digital Sky Survey (SDSS) dataset. They extract the terms in *selection*, *joins*, *projection*, *from*, *group-by* and *order-by* items separately and record their appearance frequency. They create a feature vector using the frequency of these terms which they use to calculate the pairwise similarity of queries with cosine similarity. Instead of clustering, they perform the workload analysis with Self-Organizing Maps (SOM).

To further illustrate how the three structural metrics [13], [14], [15] work, we show the feature representations for the following query for each method in Table 2.

```
SELECT u.username, u.yearenrolled
FROM user u, accounts a
WHERE u.id = a.userid
  AND a.balance > 1000
  AND u.id > 20050001
GROUP BY u.yearenrolled
ORDER BY u.yearenrolled
```

In the next section, we propose a generalized feature engineering scheme for query comparison methods to improve the clustering quality. Our work evaluates the performance of the three methods [13], [14], [15] that directly describe a pairwise similarity metric in Section 4 due to the lack of performance evaluation for the query similarity metrics in the given studies. We also show that our feature engineering scheme improves the clustering quality with both statistical and empirical methods.

3 Feature Engineering

The grammar of SQL is declarative. By design, users can write queries in the way they feel most comfortable, letting well-established equivalence rules dictate a final evaluation strategy. As a result, many syntactically distinct queries may still be semantically equivalent. Recall example queries 1 and 3, paraphrased here:

```
1) SELECT name FROM user
   WHERE rank = 'a' OR rank='s'
3) SELECT name FROM user WHERE rank = 'a'
   UNION SELECT name FROM user WHERE rank = 's'
```

Though semantically distinct, these queries produce identical results for any input. Unfortunately similarity of results is not practical to implement: General query equivalence is NP-complete [22] for SQL92 and earlier, while SQL99 and later versions of SQL are turing-complete, due to the introduction of recursive queries.

However, we can still significantly improve clustering quality by standardizing certain SQL features into a more regular form with techniques such as canonicalizing names and aliases, removing syntactic sugaring, and standardizing nested query predicates. This process of *regularization* aims to produce a new query that is more likely to be *structurally* similar to other *semantically* similar queries. Because the output is an ordinary SQL query, regularization may be used with any similarity metric. These process is similarly used in [23], [24], where Chandra *et al.* [23] generate mutations of SQL queries to catch diversions from a baseline query, and Sapia [24] creates OLAP query prototypes based on selected features and models user profiles.

Although the techniques we utilize for regularization are widely used in other settings, to the best of our knowledge, we introduce their usage to improve clustering quality. We also test all the techniques we use individually to find their impact on the regularization's overall effect. Our experiments in Section 5.2 show consistent improvements for all metrics evaluated in practical real world settings. In this section, we describe the transformations that we apply to regularize queries and the conditions under which they may be applied.

3.1 Regularization Rules

Canonicalize Names and Aliases. As we will show in our experiments in Section 5, table and attribute aliases are a significant source of error in matching. Consider the following two queries:

```
5) SELECT name FROM user
6) SELECT id
   FROM (SELECT name AS id FROM user) AS t
```

Although these queries are functionally identical, variable names are aliased in different ways. This is especially damaging for the three structural heuristics that we evaluate, each of which assumes that variable names follow a globally consistent pattern. Our first regularization step attempts to create a canonical naming scheme for both attributes and tables which is similar to one used in [23].

Syntax Desugaring. We remove SQL's redundant syntactic sugar following basic pattern-replacements as shown in Table 3.

Before	After
$b\ \{>, \geq\}\ a$	$a\ \{<, \leq\}\ b$
x **BETWEEN** (a,b)	$a \leq x$ **AND** $x \leq b$
x **IN** (a,b,. . .)	x=a **OR** x=b **OR** . . .
isnull(x,y)	**CASE WHEN** x is null **THEN** y **END**

TABLE 3: Syntactic Desugaring

EXISTS Standardization. Although SQL admits four classes of nested query predicates: (`EXISTS`, `IN`, `ANY`, and `ALL`), the `EXISTS` predicate is general enough to capture the semantics of the remaining operators [23]. Queries using the others are rewritten:

`x IN (SELECT y ...)` **becomes**
`EXISTS (SELECT * ...WHERE x = y)`
`x < ANY (SELECT y ...)` **becomes**
`EXISTS (SELECT * ...WHERE x < y)`
`x < ALL (SELECT y ...)` **becomes**
`NOT EXISTS (SELECT * ...WHERE x ≥ y)`

DNF Normalization. We normalize all boolean-valued expressions by converting them to disjunctive normal form (DNF). The choice of DNF is motivated by the ubiquity of conjunctive queries in most database applications, as well as by the natural correspondence between disjunctions and unions that we exploit below.

Commutative Operator Ordering. We standardize the order of expressions involving commutative and associative operators (e.g., $\wedge$, $\vee$, $+$, and $\times$) by defining a canonical order of all operands and traversing the expression tree bottom-up to ensure consistent order of all operands.

Flatten FROM-Nesting. We merge nested sub-queries in a `FROM` clause with its parent query as described in [23].

Nested Query De-correlation. A common database optimization called nested-query de-correlation [25] converts some `EXISTS` predicates into joins for more efficient evaluation. Note that this rewrite does not guarantee query result equivalence under *bag semantics* due to duplicated rows in the result. Hence we require that the parent query is either a `SELECT DISTINCT` or a duplicate-insensitive aggregate [26] (e.g. $max\{1,1\} = max\{1\}$, but $sum\{1,1\} \neq sum\{1\}$). If the `EXISTS` predicate is in a purely conjunctive `WHERE` clause, the de-correlation process simply moves the query nested in the `EXISTS` into the `FROM` clause of its parent query. The (formerly) nested query's `WHERE` clause can be then merged into the parent's `WHERE` clause. Specifically, if the input query is of the form:

```
SELECT ... FROM R WHERE
           EXISTS (SELECT ... FROM S WHERE q)
```

then the output query will have the form:

```
SELECT ... FROM R, (SELECT ... FROM S) WHERE q
```

To de-correlate a `NOT EXISTS` predicate, we use the set-difference operator `EXCEPT`. If the input is of the form:

```
SELECT DISTINCT... FROM R WHERE
       NOT EXISTS (SELECT ... FROM S WHERE q)
```

then the output will be of the form

```
(SELECT DISTINCT... FROM R) EXCEPT
  (SELECT DISTINCT... FROM R, WHERE
           EXISTS (SELECT ... FROM S WHERE q))
```

OR-UNION Transform. We use a regularization transformation that exploits the relationship between OR and UNION. This rewrite does not guarantee query result equivalence, also due to potentially duplicated rows in query result. Recall the equivalence between logical OR and UNION mentioned in our first example. Naively, we might convert the DNF-form predicates into UNION

queries:
`SELECT ... WHERE q OR p OR ...` **becomes**
`SELECT ... WHERE q UNION SELECT ... WHERE p UNION ...`
However, duplicates caused by the possible correlation between clauses in DNF will break the equivalence of this rewrite. Consider the following query:

```
SELECT Score FROM Exam WHERE Score>60 OR Pass=1
```

Students who pass the exam overlap with those whose score greater than 60. Thus the rewritten query would not be exactly equivalent, as it may include duplicate rows. As a result, we require the query to satisfy the same condition mentioned in previous rule *nested query de-correlation*.

Union Pull-Out. Since the prior transformation may introduce UNION operator in nested subqueries, we push selection predicates down into the union as well.

4 Quality Metrics

In this section, we introduce the quality measures and workloads to evaluate three query similarity metrics and the feature engineering scheme. Our goal is to evaluate how well a query similarity metric captures the task behind a query with and without regularization. We use two types of real-world query workloads: human- and machine-generated. We expect the problem of query similarity to be harder on human-generated workloads, as queries generated by machines are more likely to follow a strict, rigid structural pattern.

As a source of human-generated queries, we use two different sets of student answers to database course assignments. Many database courses include homework or exam questions where students are asked to translate prose into a precise SQL query. This provides us with a ground-truth source of queries with different structures that should be similar. As machine-generated queries, we use PocketData [2] a log of 33 million queries issued by smartphone apps running on 11 phones in the wild over the course of a month.

In subsection 4.1, we outline the datasets used. Then, in subsection 4.2, we outline the experimental methodology used to evaluate distance metrics, and propose a set of measures for quantitatively assessing how effective a query similarity metric is at clustering queries with similar tasks.

4.1 Workloads

We use three specific query sets: Student assignments gathered by IIT Bombay [16], student exams gathered at our department (denoted as UB dataset in the experiments) and released as part of this article[1], and SQL query logs of the Google+ app extracted from PocketData dataset [2].

The first dataset [16] consists of student answers to SQL questions given in IIT Bombay's undergraduate databases course. The dataset consists of student answers to 14 separate query-writing tasks, given as part of 3 separate homework assignments. The query writing tasks have varying degrees of difficulty. Answers are not linked to anonymous student identifiers and there is no grade information. The IIT Bombay dataset is exclusively answers to homework assignments, so we expect generally high-quality answers due to the lack of time pressure and availability of resources for validating query correctness.

The second dataset consists of student answers to SQL questions given as part of our department's graduate database course. The dataset consists of student answers to 2 separate query-writing tasks, each given as part of midterm exams in 2014 and 2015 respectively. SQL queries were transcribed from hand-written exam answers, anonymized for IRB compliance and labeled with the grade the answer was given. We expect quality to vary, as exams are closed-book and students have limited time. Since 50% of the grade is the failing criterion, we assume that answers conform with the task of the question if the grade is over 50%. We also explore 20% and 80% thresholds in Appendix B.

The third dataset consists of SQL logs that capture all database activities of 11 Android phones for a period of one month. We selected Google+ application for our study since it is one of the few applications where all users created a workload. SQL queries collected were anonymized and some of the identified query constraints were deleted for IRB compliance [2].

A summary of all datasets is given in Tables 4, 5, and 6. The prose questions asked for IIT Bombay and UB Exam datatsets can be found in Table 7 and 8. Not all student responses are legitimate SQL, and so we ignore queries that cannot be successfully parsed by our open-source SQL parser[2]. We also released the source code we used in the experiments[3].

In the first two datasets, the query-writing task is specific. We can expect that student answers to a single question are written with the same task. Thus, we would expect a good distance metric to rate answers to the same question as close and answers to different questions as distant. Similarly, using the distance metric for clustering, we would expect to see each query cluster to uniformly include answers to the same question.

In the third dataset, PocketData-Google+, the queries are generated by the Google+ application. Since some of the constants are replaced with standard placeholders for IRB compliance, the number of distinct queries drops significantly. Since there is no information about what kind of a task a query is trying to perform, we inspected and manually labeled each distinct query string. Queries were labeled with one of 8 different categories: Account, Activity, Analytics, Contacts, Feed, Housekeeping, Media and Photo.

4.2 Clustering validation measures

In addition to workload datasets, we define a set of measures to be used for evaluating queries. Given a set of queries labeled with tasks and an inter-query similarity metric, we want to understand how well the metric can (1) put queries that perform the same task close together even if they are written differently, and (2) differentiate queries that are labeled with different tasks.

We evaluate each metric according to how well it aligns with the ground-truth cluster labels. Rather than evaluating the clustering output itself, we evaluate an intermediate

[1] http://odin.cse.buffalo.edu/public_data/2016-UB-Exam-Queries.zip

[2] https://github.com/UBOdin/jsqlparser

[3] https://github.com/UBOdin/EttuBench

Question	Total number of queries	Number of parsable queries	Number of distinct query strings
1	55	54	4
2	57	57	10
3	71	71	66
4	78	78	51
5	72	72	67
6	61	61	11
7	77	66	61
8	79	73	64
9	80	77	70
10	74	74	52
11	69	69	31
12	70	60	22
13	72	70	68
14	67	52	52

TABLE 4: Summary of IIT Bombay dataset

Year	2014	2015
Total number of queries	117	60
Number of syntactically correct queries	110	51
Number of distinct query strings	110	51
Number of queries with score $>$ 50%	62	40

TABLE 5: Summary of UB Exam dataset

ID	Question
1	Find course_id and title of all the courses
2	Find course_id and title of all the courses offered by “Comp. Sci.” department.
3	Find course_id, title and instructor ID for all the courses offered in Spring 2010
4	Find id and name of all the students who have taken the course “CS-101”
5	Find which all departments are offering courses in Spring 2010
6	Find the course ID and titles of all courses that have more than 3 credits
7	Find, for each course, the number of distinct students who have taken the course; in case the course has not been taken by any student, the value should be 0
8	Find id and title of all the courses offered in Spring 2010, which have no pre-requisite
9	Find the ID and names of all students who have (in any year/semester) taken two courses
10	Find the departments (without duplicates) of courses that have the maximum credits
11	Show a list of all instructors (ID and name) along with the course_id of courses they have taught. If they have not taught any course show the ID and name with null value for course_id
12	Find IDs and names all students whose name contains the substring “sr” ignoring case. (Hint Oracle supports the functions lower and upper)
13	Using a combination of outer join and the is null predicate but WITHOUT USING ”except/minus” and ”not in” find IDs and names of all students who have not enrolled in any course in Spring 2010
14	A course is included in your CPI calculation if you passed it, or you have failed it, and have not subsequently passed it (or in other words, a failed course is removed from CPI calculation if you have subsequently passed it). Write an SQL query that shows all tuples of the relation other than those eliminated by the above rule, and also eliminating tuples with a null value for grade

TABLE 7: Questions given IIT Bombay Dataset [16]

Year	Question
2014	How many distinct species of bird have ever been seen by the observer who saw the most birds on December 15, 2013?
2015	You are hired by a local birdwatching organization, who’s database uses the Birdwatcher Schema on page 2. You are asked to design a leader board for each species of Bird. The leader board ranks Observers by the number of Sightings for Birds of the given species. Write a query that computes the set of names of all Observers who are highest ranked on at least one leader board. Assume that there is no tied rankings.

TABLE 8: UB Exam dataset questions

step: the pairwise distance matrix for the set of queries in a given workload. With this matrix and a labeled dataset, we can use various clustering validation measures to understand how effectively a similarity metric characterizes the partition of a set of queries. Specifically, clustering validation measures are used to validate the quality of a labeled dataset by estimating two quantities: (1) the degree of tightness of observations in the same label group and (2) the degree of separations between observations in different label groups. As a result, we will use three clustering validation measures [17, Chapter 17] including Average Silhouette Coefficient, BetaCV and Dunn Index as they all quantify the two qualities mentioned above in their formulations.

Silhouette coefficient. For every data point in the dataset, its silhouette coefficent is a measure of how similar it is to its own cluster in comparison to other clusters. In particular, the silhouette coefficient for a data point i is measured as $\frac{b(i)-a(i)}{max(a(i),b(i))}$ where $a(i)$ is the average distance from i to all other data points in the same cluster and $b(i)$ is the average distance from i to all other data points in the closest neighboring cluster. The range of silhouette coefficient is from -1 to 1. We denote $s(i)$ to represent silhouette coefficient of data point i. $s(i)$ is close to 1 when $s(i)$ is close to other data points from the same cluster more than data points from different clusters, which represents a good match. On the other hand, $s(i)$ which is close to -1 represents that the data point i stayed in the wrong cluster, as it is closer to data points in different clusters than its own. Since the silhouette coefficient represents a measure of degree of goodness for each data point, to validate the effectiveness of the distance metric given a query partition, we use the average silhouette coefficient of all data points (all queries) in the dataset.

	Pocket Dataset	Google+
All queries	45,090,798	2,340,625
SELECT queries	33,470,310	1,352,202
Distinct query strings	34,977	135

TABLE 6: Summary of PocketData dataset and Google+

BetaCV measure. The BetaCV measure is the ratio of the total mean of intra-cluster distance to the total mean of inter-cluster distance. The smaller the value of BetaCV, the better the similarity metric characterizes the cluster partition of queries on average.

Dunn Index. The Dunn Index is defined as the ratio between minimum distance between query pairs from different clusters and the maximum distance between query pairs from the same cluster. In other words, this is the ratio between closest pairs of points from different clusters over the largest diameter among all clusters. Higher values of the Dunn Index indicate better the worst-case performance of the clustering metric.

5 Experiments

In this section, we perform experiments to evaluate the performance of three similarity metrics previously discussed in Section 2: Makiyama's similarity [15], Aligon's similarity [14] and Aouiche's similarity [13]. We implemented each of these similarity metrics in Java and evaluated them using the three clustering validation measures discussed in subsection 4.2. In particular, we evaluate these three similarity metrics on their ability to capture the tasks performed by SQL queries. In addition, we also evaluate the effectiveness of the feature engineering step introduced in Section 3 and understand how query similarity can be improved by applying this step on the SQL query. We also look closer at feature engineering by breaking it down to different modules and analyze the effect of each module on capturing the tasks performed by queries.

5.1 Evaluation on SQL similarity metrics

In the first experiment, we evaluate three similarity metrics mentioned in Section 2. The aim of the experiment is to evaluate which similarity metric can best capture the task performed by each query.

The black columns in Figure 1 show a comparison of three similarity metrics using each of the three quality measures (Average Silhouette Coefficient, BetaCV and Dunn Index). As can be seen in Figure 1, Aligon seems to work the best for both IIT Bombay and UB Exam dataset while achieving second-best for PocketData-Google+ dataset under the Average Silhouette Coefficient measure. When considering BetaCV measure, Aligon also attains the best result for both IIT Bombay and UB Exam dataset while having comparable result for PocketData-Google+ dataset. Aligon also performs well on the Dunn Index, coming in first on UB Exam dataset, and second-best for IIT Bombay and PocketData-Google+ dataset. Especially given that the Dunn Index measures only worst-case performance, Aligon's metric seems to be ideal for our workloads. This shows that even a fairly simple approach can capture task similarity well.

For a closer look of Aligon's similarity metric, Figure 2(a,c,e) shows the distribution of Silhouette coefficients for each query and their respective tasks. Recall that the silhouette coefficient below 0 effectively indicates a query closer to another cluster than its own, or a query that would be mis-classified. The further below zero, the greater the error. For the UB Exam dataset (Figure 2c), the majority of queries would have been successfully classified, and only a small fraction exhibit minor errors. For the PocketData-Google+ dataset (Figure 2e), there are some erroneous queries in cluster 4, 5 and 6 while cluster 1, 2, 3, 7 and 8 have very few errors. For the Bombay dataset (Figure 2a), the distribution of errors varies. Cluster 1, 2, 4, 6, 12 and 14 exhibit virtually no error, while cluster 7, 8, and 9 exhibit particularly egregious errors.

5.2 Evaluation of feature engineering

We next evaluate the effectiveness of regularization by applying it to each of the three metrics described in Section 2. We use our quality evaluation scheme to compare the quality of each measure both with and without feature engineering.

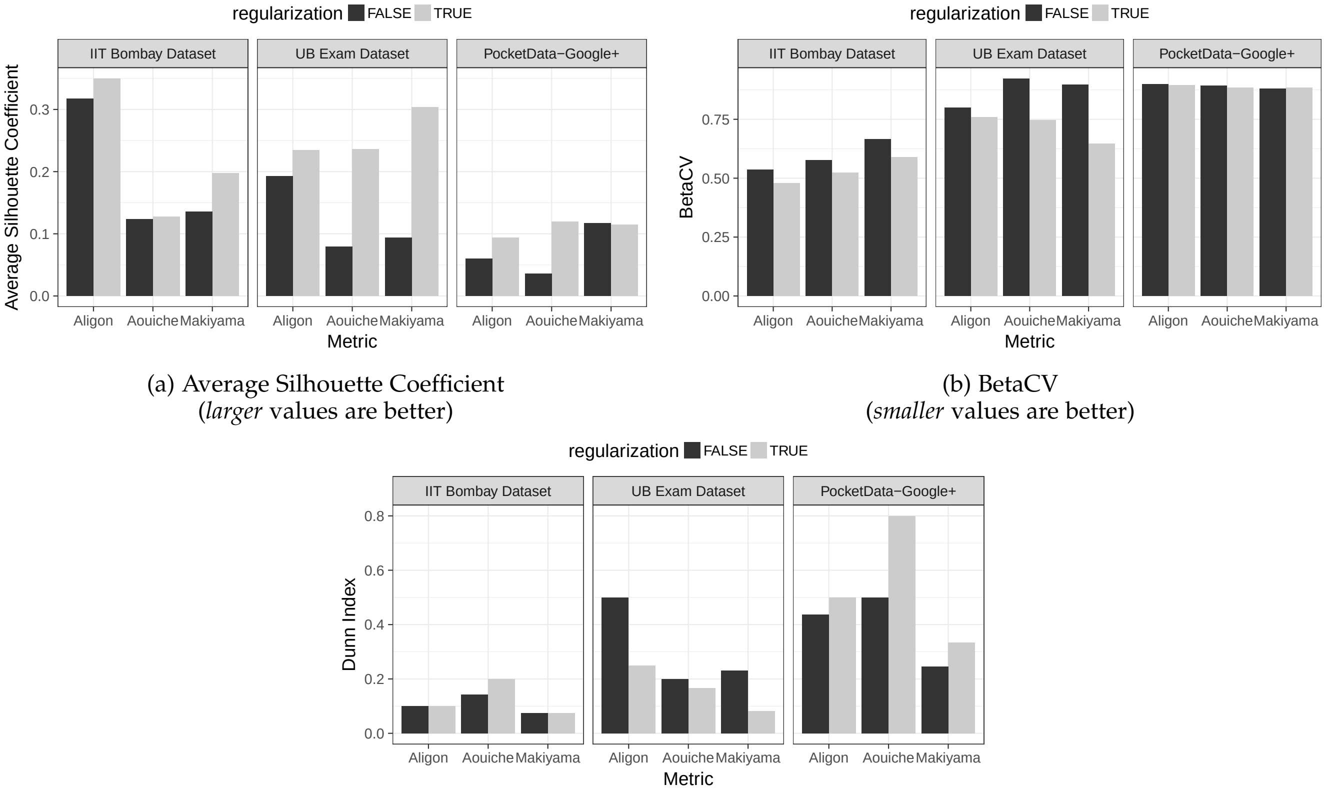
Figure 1 shows the values of three validation measures for each of the three similarity metrics, both with and without regularization. As shown in Figure 1, regularization significantly improves the Average Silhouette Coefficient and BetaCV measures for all similarity metrics except for the case of Makiyama similarity metric with PocketData-Google+ dataset. The Dunn index is relatively unchanged or little improved for the IIT Bombay and PocketData-Google+ dataset and shows slight signs of worsening with regularization on the UB Exam dataset. To understand the reason of worse Dunn Index, we compare Figure 2c (original) with Figure 2d (with regularization). The Silhouette Coefficient for answers that are originally positive in each question are considerably increased, and for answers that are originally negative (regarded erroneous) are even more decreased as a result of regularization, since it reduces the query structure diversity which leads to separating queries better. In other words, for erroneous answers with negative Silhouette Coefficients, distance metrics like Aligon distinguish them further apart from answers with positive Silhouette Coefficients after regularization. Since erroneous answers are treated as the 'worst cases' for each question, the Dunn Index which measures worst case performance naturally gets worse.

5.2.1 *Per-Query Similarity*

Figure 2(b,d,f) shows the distributions of silhouette coefficients for the Aligon similarity metric after regularization is applied. For IIT Bombay dataset, comparing against Figure 2a there is a slight improvement at the tail end of clusters 9, 11, 12, 13 and 14 — several of the negative coefficients have been removed. Furthermore, positive matches have been improved, particularly for cluster 7, 9, 10, 12 and 13. Finally, there has been a significant reduction in the degree of error in cluster 10. Cluster 10 is a particularly egregious case of aliasing, as the correct answer involves two self-joins in the same query. As a result, aliasing is a fundamental part of the correct query answer, and our rewrites could not reliably create a uniform set of alias names. In the UB Exam and PocketData-Google+ datasets, the improvement provided by regularization can be seen for queries with both positive and with negative values of $s(i)$.

5.3 Case Study

As part of our analysis, we attempted to provide empirical explanations for query errors, in particular for queries where $s(i) < 0$ for all three similarity metrics. Namely, we looked into the queries that are too far apart from the clusters they belong, and we categorized the reasons for misclassification



(a) Average Silhouette Coefficient
(*larger* values are better)

(b) BetaCV
(*smaller* values are better)

(c) Dunn Index
(*larger* values are better)

Fig. 1: Clustering validation measures for each metric with and without regularization step

based on these queries. We then investigated how the regularization process particularly affect these queries.

Almost all of these egregiously misclassified queries appear in the IIT Bombay dataset, the distribution of which is summarized in Table 9. The PocketData-Google+ dataset includes no egregiously misclassified queries, while the UB Exam dataset includes only one such query (which we tagged as a case of **Contextual equivalence**). We tagged each egregiously misclassified query with an explanation that justifies why the query has a low s(i). Tags were drawn from the following list:

Ground-truth error. A student's response to the question may have been legitimately incorrect. This is a query that is correctly classified as an outlier. For example:

```
SELECT *
FROM (SELECT id, name, time_slot_id
 FROM (SELECT *
  FROM (SELECT *
   FROM student
    NATURAL JOIN takes) b1) a, section
   WHERE a.course_id = section.course_id) a1
```

This query was attempting to complete the task *"Find the ID and names of all students who have (in any year/semester) taken two courses in the same timeslot."*

Nested subquery. A student's response is equivalent to a legitimately correct answer but uses nested subqueries such that a heuristic distance metric cannot recognize. For example:

```
SELECT id, name FROM student
WHERE id IN (SELECT DISTINCT s.id
 FROM (SELECT * FROM takes NATURAL JOIN section) s,
  (SELECT * FROM takes NATURAL JOIN section) t
 WHERE s.id = t.id
  AND s.time_slot_id = t.time_slot_id
  AND s.course_id <> t.course_id)
```

Here, the subquery nesting structure is significantly different from other queries for of the same question.

Aliasing. Aliasing (e.g., AS in SQL) breaks a distance metric that relies on attribute and relation names. For example:

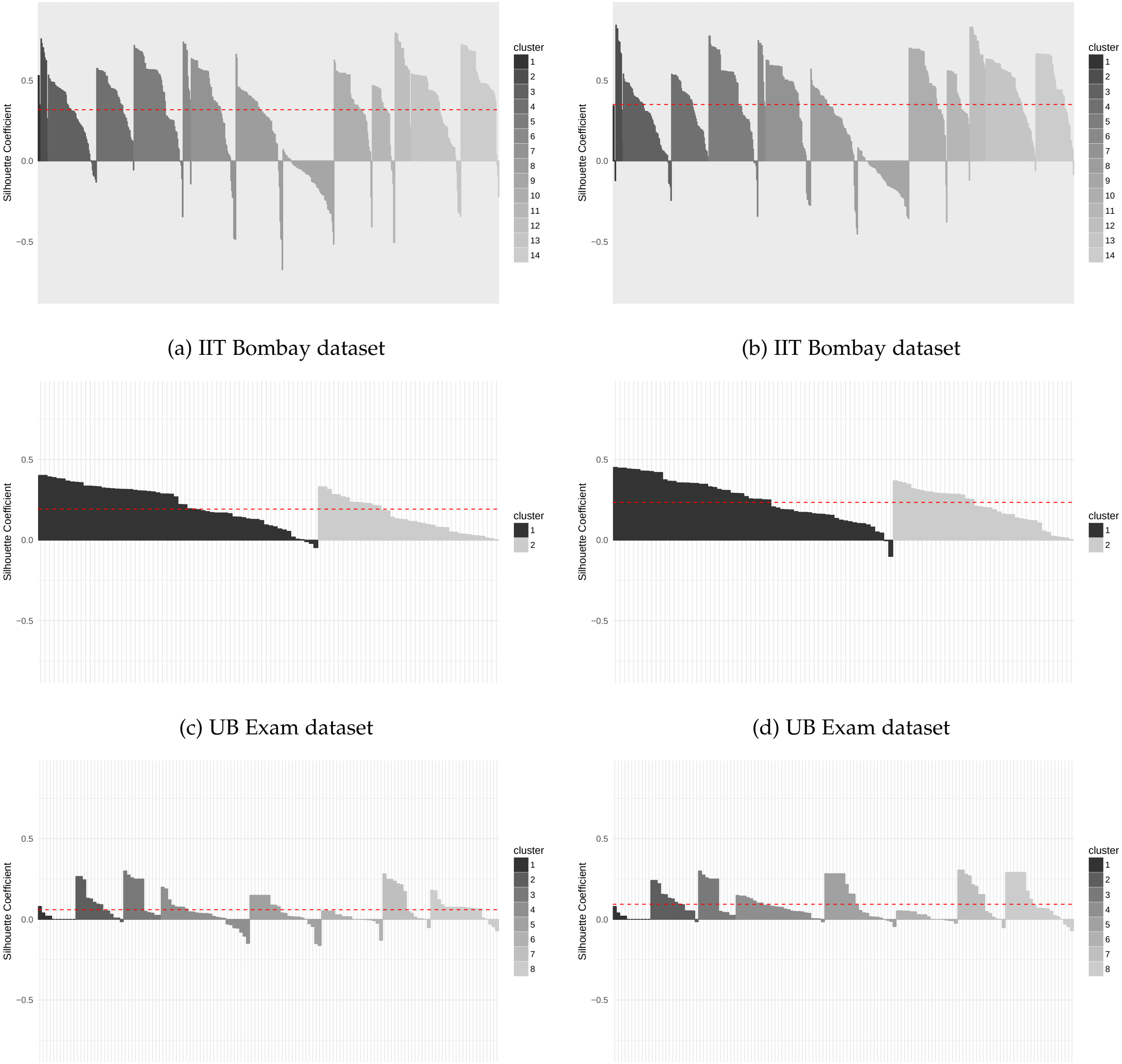
```
SELECT DISTINCT student.id, student.name
FROM student, takes, section AS a, section AS b
WHERE student.id = takes.id
  AND takes.course_id = a.course_id
  AND takes.course_id = b.course_id
  AND a.course_id <> b.course_id
  AND a.time_slot_id = b.time_slot_id
```

The student's use of a and b make this query hard to distinguish from other queries that may use other names for the attributes.

Insufficient features. Relevant query components are not sufficiently captured as features for a heuristic distance metric to distinguish between answers from sufficiently similar questions.

Too many features. Irrelevant query components create redundant features that artificially increase the distance between the query and cluster center. For example:

```
SELECT DISTINCT student.name, takes.id,
                s1.course_id, s2.course_id
FROM section AS s1, section AS s2, takes, student
WHERE takes.course_id = s1.course_id
  AND s1.course_id <> s2.course_id
```



(a) IIT Bombay dataset

(b) IIT Bombay dataset

(c) UB Exam dataset

(d) UB Exam dataset

(e) PocketData-Google+ dataset

(f) PocketData-Google+ dataset

Fig. 2: Distribution of silhouette coefficients when using Aligon's similarity without regularization (a,c,e), and when regularization is applied (b,d,f)

```
  AND s1.time_slot_id = s2.time_slot_id
  AND s1.semester = s2.semester
  AND s1.year = s2.year
  AND takes.sec_id = s1.sec_id
  AND s1.semester = takes.semester
  AND s1.year = takes.year
  AND student.id = takes.id
  AND s2.time_slot_id = s2.time_slot_id
  AND takes.sec_id = s2.sec_id
  AND s2.semester = takes.semester
  AND s2.year = takes.year
```

Contextual equivalence. Establishing query equivalence to properly clustered queries requires domain-specific knowledge not available to the distance metric (e.g. attribute uniqueness). For example:

```
SELECT student.id, student.name
FROM student
WHERE student.id
  IN (SELECT takes.id
    FROM takes, section
    WHERE takes.course_id = section.course_id
      AND takes.sec_id = section.sec_id
      AND takes.semester = section.semester
      AND takes.year = section.year
    GROUP BY takes.id,
        takes.semester,
        takes.year,
        section.time_slot_id
    HAVING count(*) > 1)
```

Table 9 shows the primary reasons why these queries could not be classified correctly. Note that there may be more than one reason for a query to be placed in a different cluster, but in Table 9, we only give the empirically determined primary reason.

Cause	Erroneous queries Without Regularization	Erroneous queries With Regularization
All queries	33 (100%)	27 (100%)
Ground-truth quality	14 (42.4%)	14 (51.8%)
Nested subquery	7 (21.2%)	5 (18.5%)
Aliasing	8 (24.2%)	5 (18.5%)
Insufficient features	2 (6.0%)	1 (3.7%)
Too many features	1 (3.0%)	1 (3.7%)
Contextual equivalence	1 (3.0%)	1 (3.7%)

TABLE 9: Empirical error reasons for IITBombay Dataset

Many of the queries with low silhouette coefficients are identified as incorrect answers for the task given. These answers directly affect the ground-truth quality, therefore reduce the average silhouette coefficient. Another reason for erroneous queries with low silhouette coefficients is because of aliasing. Although it is convenient for user to use aliases in the query to refer to a particular item, it is difficult for a machine to approximate the tasks the query authors are trying to accomplish since different query authors have different ways to name particular items in the query. This problem is particularly prevalent in question 9 of the IIT Bombay dataset.

Although the distribution of the error reasons are expected to change, all the tags provided in this section can generically be applied to other query logs given a ground-truth. The regularization method cannot be expected to fix errors originating from misclassifications in ground-truth since they do not actually share any similarities with the cluster.

After the regularization process, the silhouette coefficient under all three similarity metrics for each query is computed again and the result yields an 18% overall reduction in number of erroneous queries ($s(i) < 0$) in the IIT Bombay dataset.

5.4 Analysis of regularization by module

In Subsection 5.2, we analyzed the overall effect of regularization on query similarity. However, as described in Section 3, regularization is composed of many different transformation rules. In this experiment, we group these rules into four separate modules and inspect their impact on the clustering quality. One may observe that Commutative Operator Ordering is guaranteed to provide benefit in structure similarity comparison, hence we include it in all four modules. In addition, there are dependencies between rules that require them to operate one before another. For example, we should better apply Syntax Desugaring and then DNF Normalization to simplify the boolean expression in WHERE clause before OR-Union Transformation. As another example, Exists Standardization should better be applied on nested sub-queries before we de-correlate them using Nested Query De-correlation. As a result, we group the rules from Section 3 into four modules:

1) *Naming*: **Canonicalize Names and Aliases**
2) *Expression Standardization*: **Syntax Desugaring**, **Exists Standardization**, **DNF Normalization**, **Nested Query Decorrelation**, **OR-Union Transform**
3) *FROM-Nesting*: **Flatten FROM-Nesting**
4) *Union Pullout*: **OR-UNION Pullout**

Commutative Operator Ordering is included in all modules.

Figure 3 provides a comparison of each module in regularization. From this figure, one can observe that, since students use different names/aliases for their convenience when constructing queries, the *Naming* module is the most effective one in terms of improving clustering quality for IIT Bombay and UB Exam datasets. On the other hand, for PocketData-Google+ dataset, names are already canonicalized as they are machine-generated. In this case, *Expression Standardization* seems to be the most effective module, especially when using Aligon or Aouiche as similarity metric. In PocketData-Google+ dataset, referred tables and boolean expressions in the queries are both informative in distinguishing between different query categories or clusters. For this reason, Makiyama similarity metric which considers both works well even without regularization while Aligon and Aouiche can get commensurate performance only after applying *Expression Standardization* module.

Note that in Figure 3, *Expression Standardization* makes Average Silhouette Coefficient worse in some cases for IIT Bombay and UB Exam data sets. The performance degradation is majorly due to *feature duplication*. More specifically, consider the example query with *Expression Standardization*.

Example 1. Syntax Desugaring with OR-UNION transform

```
1) SELECT name FROM usr WHERE
   rank IN {’admin’,’normal’}
2) SELECT name FROM usr WHERE
   rank = ’admin’ OR rank = ’normal’
3) SELECT name FROM usr
   WHERE rank = ’admin’
   UNION
   SELECT name FROM usr
   WHERE rank = ’normal’
```

Query (1) is transformed into (2) by syntax desugaring and then into (3) by OR-UNION Transform. From (1) to (2), feature `WHERE rank` has been replicated; From (2) to (3), features `SELECT name` and `FROM usr` have been duplicated. For expressions of the form: `X IN` $\{x_1, x_2, \ldots, x_n\}$, feature duplication becomes dominant when n grows large. In Figure 3, Aligon and Makiyama suffer from feature duplication brought by *Expression Standardization* in some cases while Aouiche does not. Because Aouiche records feature existence instead of occurrence in its vector. Although in some cases such as this, simply replacing feature occurence with existence solves the problem of feature duplication, feature occurence can also be a good indicator for the interests of the query. We believe this problem can be addressed with exploration of feature weighting strategies. Therefore, the problem of feature duplication will be further explored as a part of feature weighting strategies in our future work.

6 Discussion

We have reviewed several similarity metrics for clustering queries and focused on three syntax-based methods that offer an end-to-end similarity metric. The advantage of this preference is that, syntax-based methods do not require access to the data in the database or database properties. Considering that only logs are usually transferred between

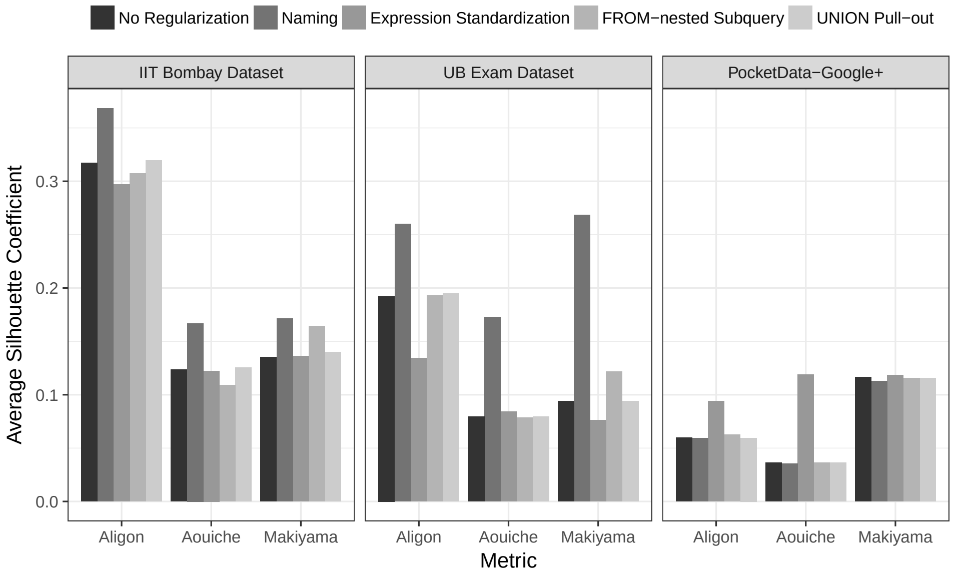


Fig. 3: Effect of each module in regularization

organizations, and requiring access to the data for investigations can cause privacy violations, we preferred focusing on the syntax-based approach.

The survey we performed shows that most of the metrics make use of selection and join operations in the queries and consider them as the most important items for similarity calculation. Group-by aggregate follows them closely while projection items take the third most important item set. There are other possible feature sets that can be used, such as tables accessed or the abstract syntax tree (AST) of a query, but these feature sets are generally overlooked.

Although Aouiche *et al.* [13] make use of the most important features selection, joins, and group-by items, they don't utilize the number of times an item appears, or after the parsing, they don't consider what kind of feature an item is. This means, it does not matter if a query has `rank` column in group-by, and the other one has `rank` column in selection; they are considered the same. Makiyama *et al.* [15], on the other hand, follow Aligon *et al.* [14] in separating the different features, and improves on it by making use of appearance count of items. However, while trying to make use of every item like `FROM` and `Order-By` predicates, they consider these low priority predicates with same importance as the selection and join predicates.

Makiyama *et al.* [15] use a more complete structure of the query AST, hence when the query is simple like in the PocketData-Google+ dataset, this technique can be slightly better. However, for a complex query with redundant features, mixing features captured from various components of a query without proper feature re-weighting will essentially decrease the weight of features that are more informative. Hence, in student exam datasets, we can observe that Aligon *et al.* [14] is better than the others while in PocketData-Google+ dataset, Makiyama *et al.* [15] is better.

We could further improve these methods by making use of the abstract syntax tree (AST) of a SQL statement. As a declarative language, the AST of a SQL statement acts as a proxy for the task of the query author. This suggests that a comparison of ASTs can be a meaningful metric for query similarity. For instance, we can group a query Q with other queries that have nearly (or completely) the same AST as Q. This structural definition of task has seen substantial use already, particularly in the translation of natural language queries into SQL [27]. For two SQL queries Q_1 and Q_2, one reasonable measure might be to count the number of connected subgraphs of Q_1 that are isomorphic to a subgraph of Q_2. Subgraph isomorphism is NP-complete, but a computationally tractable simplification of this metric can be found in the Weisfeiler-Lehman (WL) Algorithm [3], [28].

As can be seen in Tables 4 and 5, as the complexity or difficulty of the question increases, the number of distinct queries also increases, i.e., students find different ways to solve the same problem. Especially, in Table 5, no two students answer a question using the same structure. This phenomenon motivates the need for regularization in comparing SQL queries. As the complexity of the query increases, the possible ways to create the query to achieve the same task increase. Figure 1 shows that our assumption that regularizing queries will improve overall clustering quality is correct. Our proposed feature engineering scheme improves the overall clustering quality of all three metrics on all three datasets, including both human- and machine-generated queries.

7 Application Scenarios

In this section, we provide three scenarios where the clustering scheme coupled with the proposed regularization is applicable:

The first one is, *Jane the DBA* where she takes on the task of improving database performance. After performing the straightforward database indexing tasks, she would need to select candidate *views*, which are virtual tables defined by a query. They allow querying just like tables by pre-fetching records from existing tables. Constructing a view for a frequent complex join operation can increase querying performance of the database substantially. To find the ideal views, Jane first clusters similar queries together to see what kinds of queries are more frequent. Making the most frequent complex query types faster by creating views of them could improve database performance substantially [13], [14].

The second one is, *Jane the security auditor* where she suspects that there is a person who leaks classified information from her organization. She can choose to investigate database access patterns along with other strategies which would involve query clustering [29]. After identifying the query clusters, she can partition the queries by the department or role to get the intuition about which departments and roles *normally* utilize what part of the database. She can detect the *outliers* from that behavior in order to determine the suspects for further investigation.

Lastly, *Jane the researcher* where needs to investigate the properties of the SQL query dataset that she is going to use for her research. One of the new graduate students in her team clusters the queries, and provides her with the clustering assignments of each query. She doubts the quality of the clustering performed, and wonders if the clustering operation could be performed better.

Having a *better* clustering of queries would potentially enhance the quality of her work in all of the examples given above. Also, works cited in this section [13], [14], [29], along with many others can benefit from the framework described in this article.

8 Conclusion and Future Work

The focus of this work is to understand and improve similarity metrics for SQL queries relying on query structure to be used to cluster queries. We described a quality evaluation scheme that captures the notion of query task using student answers to query-construction problems and a real-world smartphone query load. We used this scheme to evaluate three existing query similarity metrics. We also proposed a feature engineering technique for standardizing query representations. Through further experiments, we showed that different workloads have different characteristics and no one similarity metric surveyed was always good. The feature engineering steps provided an improvement across the board because they addressed the error reasons we identified.

The approaches described in this article only represent the first steps towards tools for summarizing logs by tasks. Concretely, we plan to extend our work in several directions: First, we will explore new feature extracting mechanisms like the Weisfeiler-Lehman framework [3], feature weighting strategies and new labeling rules in order to capture the task behind logged queries better. Second, we will introduce the temporal order of the log to increase the query clustering quality. In this article, we focused on query structures to improve clustering quality. Exploring the inter-query feature correlation based on query order can be used to summarize query logs in addition to clustering. Third, we will examine user interfaces that better present clusters of queries — Different feature sorting strategies in Frequent Pattern Trees (FP Trees) [30] in order to help the user distinguish important and irrelevant features, for example. Lastly, we will investigate the temporal effects on query clustering.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under award number CNS - 1409551. Usual disclaimers apply.

References

[1] N. Bruno and S. Chaudhuri, “Automatic physical database tuning: A relaxation-based approach,” in *ACM SIGMOD*, 2005.

[2] O. Kennedy, J. Ajay, G. Challen, and L. Ziarek, “Pocket Data: The need for TPC-MOBILE,” in *TPC-TC*, 2015.

[3] G. Kul, D. Luong, T. Xie, P. Coonan, V. Chandola, O. Kennedy, and S. Upadhyaya, “Ettu: Analyzing query intents in corporate databases,” in *WWW Companion*, 2016.

[4] C. Dwork, “Differential privacy,” in *Automata, Languages and Programming*, M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, Eds. Springer, 2006.

[5] C. Sapia, “Promise: Predicting query behavior to enable predictive caching strategies for OLAP systems,” in *DaWaK*, 2000.

[6] A. Giacometti, P. Marcel, E. Negre, and A. Soulet, “Query recommendations for OLAP discovery driven analysis,” in *ACM DOLAP*, 2009.

[7] X. Yang, C. M. Procopiuc, and D. Srivastava, “Recommending join queries via query log analysis,” in *IEEE ICDE*, 2009.

[8] K. Stefanidis, M. Drosou, and E. Pitoura, “”You May Also Like” results in relational databases,” in *ACM DOLAP*, 2009.

[9] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu, “SnipSuggest: context-aware autocompletion for SQL,” *pVLDB*, 2010.

[10] G. Chatzopoulou, M. Eirinaki, S. Koshy, S. Mittal, N. Polyzotis, and J. S. V. Varman, “The QueRIE system for personalized query recommendations,” *IEEE Data Eng. Bull.*, 2011.

[11] W. Gatterbauer, “Databases will visualize queries too,” *pVLDB*, 2011.

[12] J. Aligon, K. Boulil, P. Marcel, and V. Peralta, “A holistic approach to OLAP sessions composition: The falseto experience,” in *ACM DOLAP*, 2014.

[13] K. Aouiche, P.-E. Jouve, and J. Darmont, “Clustering-based materialized view selection in data warehouses,” in *ADBIS*, 2006.

[14] J. Aligon, M. Golfarelli, P. Marcel, S. Rizzi, and E. Turricchia, “Similarity measures for OLAP sessions,” *Knowledge and information systems*, 2014.

[15] V. H. Makiyama, M. J. Raddick, and R. D. Santos, “Text mining applied to SQL queries: A case study for the SDSS SkyServer,” in *SIMBig*, 2015.

[16] B. Chandra, B. Chawda, B. Kar, K. V. Reddy, S. Shah, and S. Sudarshan, “Data generation for testing and grading SQL queries,” *VLDBj*, 2015.

[17] M. J. Zaki and W. Meira Jr, *Data mining and analysis: fundamental concepts and algorithms*. Cambridge University Press, 2014.

[18] A. Kamra, E. Terzi, and E. Bertino, “Detecting anomalous access patterns in relational databases,” *VLDBJ*, 2007.

[19] S. Mathew, M. Petropoulos, H. Q. Ngo, and S. Upadhyaya, “A data-centric approach to insider attack detection in database systems,” in *RAID*, 2010.

[20] H. V. Nguyen, K. Böhm, F. Becker, B. Goldman, G. Hinkel, and E. Müller, “Identifying user interests within the data space-a case study with skyserver,” in *EDBT*, 2015.

[21] R. Agrawal, R. Rantzau, and E. Terzi, “Context-sensitive ranking,” in *ACM SIGMOD*, 2006.

[22] A. K. Chandra and P. M. Merlin, “Optimal implementation of conjunctive queries in relational data bases,” in *STOC*, 1977.

[23] B. Chandra, M. Joseph, B. Radhakrishnan, S. Acharya, and S. Sudarshan, “Partial marking for automated grading of SQL queries,” *VLDBj*, vol. 9, no. 13, pp. 1541–1544, Sep. 2016.

[24] C. Sapia, “On modeling and predicting query behavior in OLAP systems.” in *DMDW*, 1999.

[25] P. Seshadri, H. Pirahesh, and T. Y. C. Leung, “Complex query decorrelation,” in *IEEE ICDE*, 1996.

[26] A. Gupta, V. Harinarayan, and D. Quass, “Aggregate-query processing in data warehousing environments,” in *pVLDB*, 1995.

[27] F. Li and H. V. Jagadish, “Constructing an interactive natural language interface for relational databases,” *pVLDB*, 2014.

[28] N. Shervashidze, P. Schweitzer, E. J. V. Leeuwen, K. Mehlhorn, and K. M. Borgwardt, “Weisfeiler-lehman graph kernels,” *JMLR*, 2011.

[29] Y. Sun, H. Xu, E. Bertino, and C. Sun, “A data-driven evaluation for insider threats,” *Data Science and Engineering*, vol. 1, no. 2, pp. 73–85, 2016.

[30] J. Han, J. Pei, Y. Yin, and R. Mao, “Mining frequent patterns without candidate generation: A frequent-pattern tree approach,” *DMKD*, 2004.

Gokhan Kul Gokhan Kul is a PhD candidate in computer science at the University at Buffalo. His research interests include database systems and cyber-threat detection. Prior to beginning the PhD program, Gokhan received his M.S. from METU in Turkey, and worked as a software engineer there. His current work focuses on threat modeling and insider threat detection under the supervision of Dr. Shambhu Upadhyaya and Dr. Oliver Kennedy.

Duc Thanh Anh Luong Duc Thanh Anh Luong is a PhD student under supervision of Dr. Varun Chandola in the Department of Computer Science and Engineering at the University at Buffalo. Prior to joining the PhD program, he completed his B.S. degree in Computer Science at University of Science at Ho Chi Minh city, Vietnam in April 2012. His research is broadly in the field of Machine Learning and Data Mining. In particular, he develop methods for probabilistic modeling and clustering in complex data. The applications of his research works include anomaly detection, clustering patient health profiles and forecasting retail demands.

Ting Xie Ting Xie is currently a PhD candidate in CSE department of University at Buffalo. He is supervised under Dr. Oliver Kennedy, and working as a research assistant in Odin Lab of University at Buffalo. His former education includes Beijing University of Posts and Telecommunication (Bachelor), University at Pennsylvania (Master), University of Illinois at Urbana-Champaign (Master).

Varun Chandola Varun Chandola is a tenure-track Assistant Professor at University at Buffalo in the Computer Science Department and the Center for Computational and Data-Enabled Science and Engineering (CDSE). His research covers the application of data mining and machine learning to problems involving big and complex data, focusing on anomaly detection from big and complex data. Before joining UB, he was a scientist in the Computational Sciences and Engineering Division at the Oak Ridge National Laboratory. He has a PhD in Computer Science and Engineering from University of Minnesota.

Oliver Kennedy Oliver Kennedy is an Assistant Professor at the University at Buffalo. Oliver’s primary area of research is Databases, although his research interests frequently cross over into Programming Languages and Datastructures. His work focuses on self-service analytics, making messy data, schema design, and physical layout decisions more approachable. Through real-world usage metrics gathered from industry collaborations and the use of real-world testbeds, Oliver’s work aims to address the practical problems faced by data consumers everywhere.

Shambhu Upadhyaya Shambhu Upadhyaya is a professor of computer science and engineering at the State University of New York at Buffalo where he also directs the Center of Excellence in Information Systems Assurance Research and Education (CEISARE), designated by the National Security Agency. Prior to July 1998, he was a faculty member at the Electrical and Computer Engineering department. His research interests are in broad areas of information assurance, computer security and fault tolerant computing. He has authored or coauthored more than 260 articles in refereed journals and conferences in these areas. His research has been supported by the National Science Foundation, U.S. Air Force Research Laboratory, the U.S. Air Force Office of Scientific Research, DARPA, and National Security Agency. He is a senior member of the IEEE.

Appendix A
Effect of Weight in Aligon Method

As presented in section 2, Aligon's method [14] can be used to compute the similarity between SQL queries. However, Aligon's method requires user to specify the weights of different parts of SQL queries that will be used in calculation of similarity. In this method, a SQL query is divided into three parts including *projection*, *group-by* and *selection-join* (see Table 1 and Table 2 for details). In this section, we explore the effect of choosing different weight for each query part in Aligon's method.

Let us denote a tuple $(w_{proj}, w_{select-join}, w_{groupby})$ as a set of weights for projection, selection-join and group-by respectively.Totally, we consider seven different combinations of weights including $(0.2, 0.4, 0.4)$, $(0.25, 0.25, 0.5)$, $(0.25, 0.5, 0.25)$, $(0.33, 0.33, 0.33)$, $(0.4, 0.2, 0.4)$, $(0.4, 0.4, 0.2)$ and $(0.5, 0.25, 0.25)$.

Figure 4 presents the effect of different sets of weights in three datasets (see Section 4.1 for descriptions of these datasets). In this experiment, we use average Silhouette coefficient (see Section 4.2 for further details) as a measure of quality for each combination of weights. In Aligon method, we have three weights, one for *projection*, *group-by* and *selection-join* respectively, and the total of weights adding up to one. For this reason, we only present *projection* weight and *group-by* weight in Figure 4 as *selection-join* weight can be computed by from *projection* and *group-by* weights.

In Figure 4, the size (area) of the point indicates the value of average Silhouette coefficient with respect to specific combination of weights. As can be seen in this figure, the values of average Silhouette coefficient change very little when adjusting the weights of projection and group-by. For this reason, in experiments shown in Section 5, we use equal weight for each query part, i.e. $(0.33, 0.33, 0.33)$, when applying Aligon's method to different datasets.

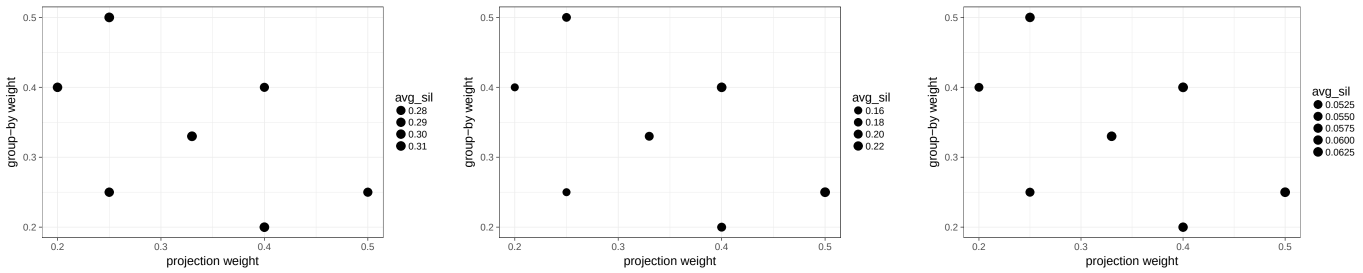
Appendix B
Exam Grade Threshold

In this section, we explore different grade cutoff thresholds in UB Exam Dataset and further investigate whether using different values of threshold can significantly affect the experimental result presented in Section 5. Table 10 gives the summary of number of queries obtained by different levels of threshold. As we can see in the table, the more demanding threshold we have, the fewer valid queries we get from the dataset.

Year	2014	2015
Total number of queries	117	60
Number of queries with score $>$ 20%	110	46
Number of queries with score $>$ 50%	62	40
Number of queries with score $>$ 80%	43	10

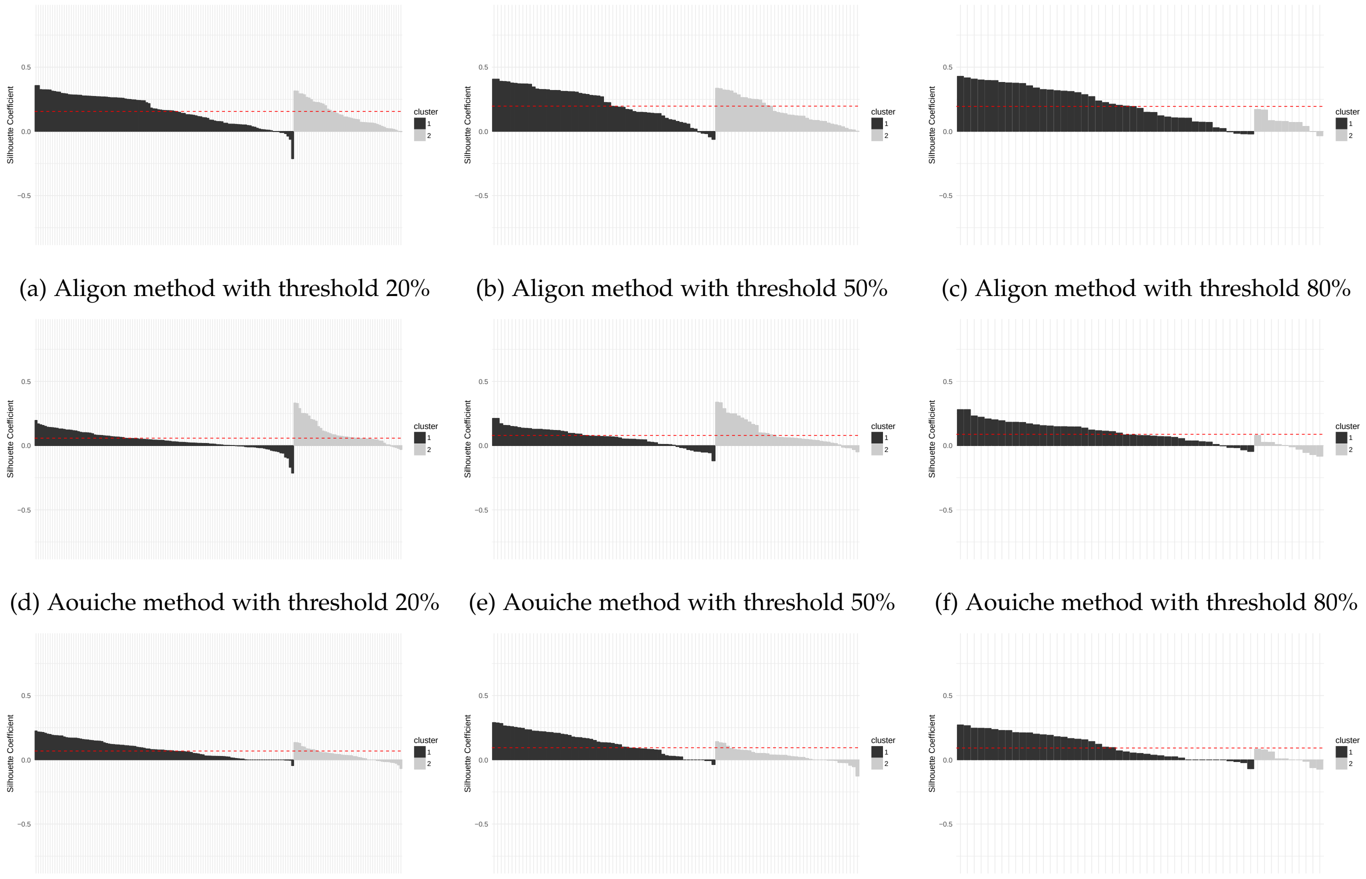
TABLE 10: Number of queries in UB Exam dataset when changing the grading threshold

In Figure 5, we show the distribution of Silhouette coefficients of each individual queries in the UB Exam Dataset and see how changing thresholds affects the quality of distance metric. As we can see in this figure, when increasing the grading thresholds, the overall values of Silhoette coeffients also improve. This observation is consistent across all three distance metrics including Aligon, Aouiche and Makiyama. However, as we mentioned earlier, increasing the grading threshold also means fewer queries for analysis. In order to balance between the number of queries and quality of analysis, we choose to use the threshold of 50% in experiments presented in Section 5.



(a) IIT Bombay Dataset (b) UB Exam Dataset (c) PocketData-Google+

Fig. 4: Avergage Silhouette Coefficients with respect to different sets of weights in Aligon method



(a) Aligon method with threshold 20% (b) Aligon method with threshold 50% (c) Aligon method with threshold 80%

(d) Aouiche method with threshold 20% (e) Aouiche method with threshold 50% (f) Aouiche method with threshold 80%

(g) Makiyama method with threshold 20% (h) Makiyama method with threshold 50% (i) Makiyama method with threshold 80%

Fig. 5: Silhouette coefficients for UB Exam Dataset when changing grading threshold in three levels - 0.2 (row 1), 0.5 (row 2), 0.8 (row 3)