Towards Effective Log Summarization *

Gokhan Kul, Duc Luong, Ting Xie,
Varun Chandola, Oliver Kennedy, Shambhu Upadhyaya
University at Buffalo, SUNY
{gokhanku, ducthanh, tingxie, chandola, okennedy, shambhu}@buffalo.edu

ABSTRACT

Database access logs are the canonical go-to resource for tasks ranging from performance tuning to security auditing. Unfortunately, they are also large, unwieldy, and it can be difficult for a human analyst to divine the intent behind typical queries in the log. With an eye towards creating tools for ad-hoc exploration of queries by intent, we analyze techniques for clustering queries by intent. Although numerous techniques have already been developed for log summarization, they target specific goals like query recommendation or storage layout optimization rather than the more fuzzy notion of query intent. In this paper, we first survey a variety of log summarization techniques, focusing on a class of approaches that use query similarity metrics. We then propose DCABench, a benchmark that evaluates how well query similarity metrics capture query intent, and use it to evaluate three similarity metrics. DCABench uses student answers to query construction assignments to capture a wide range of distinct SQL queries that all have the same intent. Next, we propose and evaluate a query regularization process that standardizes query representations, significantly improving the effectiveness of the three similarity metrics tested. Finally, we propose an entirely new similarity metric based on the Weisfeiler-Lehman (WL) approximate graph isomorphism algorithm, which identifies salient features of a graph — or in our case, of the abstract syntax tree of a query. We show experimentally that distances in WL-feature space capture a meaningful notion of similarity, while still retaining competitive performance.

Keywords

Benchmark; Query Logs; Similarity Metric; Summarization

1. INTRODUCTION

Database access logs are used in a wide variety of settings, including evaluating employee workforce allocation, database performance tuning [1], benchmark development [2], database auditing [3], and compliance validation [4]. 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 nature 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, our group was able to obtain a log of all query activity at a major US bank for over a period of 19 hours. The log includes nearly 17 million SQL queries and over 60 million stored procedure execution events. Even excluding stored procedures, it is unrealistic to expect any human to manually inspect all 17 million queries.

Let us consider an analyst (let's call her Jane) faced with the task of analyzing such a query log. She might first attempt to identify some aggregate properties about the log. 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 do not always provide a clear picture of how the data is being used, combined, and/or manipulated. To get a complete picture of the intent of users and applications interacting with the database, Jane must look at entire queries. So, she might turn to more coarse–grained properties, like the top–k most frequent queries. Here too she would run into a problem. Consider the following two queries:

```
SELECT username FROM USER WHERE rank = "admin"
SELECT username FROM USER WHERE rank = "moderator"
```

They differ only in the value of a single constant: "admin" or "moderator." When counting queries, should she count these as the same query or two different queries? If she chooses to count them together, are there perhaps other "similar" queries that she should also count together, like for example:

```
SELECT username FROM USER WHERE dept = "Finance"
```

Of course, the answer depends on the content of the log, the database schema, database records, and numerous other details that may not be available to Jane immediately when she sits down to analyze a log and group similar queries together. As a result, this type of log analysis can quickly become a tedious, time-consuming process.

Even just communicating the intent behind a single query

*The first three authors contributed equally and should be considered a joint first author

accurately remains a research challenge [5]. Thus, in this paper we focus on a specific part of the log summarization and ask "How can we group or cluster queries by their intent?" Similar questions have been studied extensively [6–14], albeit aimed at specific goals like view selection or query recommendation. For us, the motivating use case is free-form exploration, making similarity of intent a more abstract, "fuzzy" notion.

We make the notion of similarity of intent more precise through a new benchmark that we propose in this paper: The database course assignment benchmark (DCABench). A key feature of most database coursework is exam or homework questions where students are asked to translate english prose into SQL. Student-written queries are appealing for several reasons. First, queries are implicitly labeled by their ground-truth clusterings — For each question the student is attempting to accomplish a specific stated task. Second, scoring information (if available) serves as ground truth for the distance metric. Higher-rated queries should be closer to the centroid of a cluster of queries. For our evaluation, we use two sets of queries, a large and un-scored set of student queries released by IIT Bombay [15] and a smaller but scored set of queries gathered at the University at Buffalo and released as part of this publication. We use this benchmark to evaluate three existing query distance metrics from the literature [12–14].

None of these measures perform as well as desired, so we propose a pre-processing step to create more regular, uniform query representations by leveraging query equivalence rules and data partitioning operations. As we show, this process significantly improves the quality of all three distance metrics, very significantly in the case of a metric proposed by Aligon et. al. [13]. After regularization, the Aligon metric becomes very effective at identifying queries in our workload.

Finally, we present a new approach to constructing distance metric based on the observation that, since SQL is a declarative language, query intent will be reflected in the query's structure and its abstract syntax tree (AST). The AST of a query is a tree structure that captures hierarchical relationships between elements of the query. For example, as seen in Figure 4 the query AST's root node might have the query's Columns (i.e., Target, or SELECT), FROM, and WHERE clauses as children. Our approach, based on the Weisfeiler-Lehman subgraph isomorphism algorithm, defines features based on subtrees of the AST and uses the resulting feature space to cluster queries. The resulting clusters are groups of queries with significantly overlapping ASTs. Our experiments show that this distance metric is accurate, outperforms the Aligon metric in a few respects, and establishes a promising direction for future research on query intent similarity.

Concretely, the contributions of this paper are: (1) A survey of SQL query similarity metrics in existing literature, (2) DCABench, a new benchmark and evaluation methodology for query similarity metrics, (3) A regularizing preprocessing step that improves the accuracy of query similarity metrics, (4) A query similarity metric based on the Weisfeiler-Lehman (WL) approximate graph isomorphism algorithm [16] as a promising direction.

Experimental results show that our naive distance function not only mirrors intuitive notions of similarity as well as others, but has performance competitive with similar clustering techniques from the literature.

This paper is organized as follows. We start by performing a literature survey on log summarization and SQL query similarity in Section 2. In Section 3, we evaluate the accuracy and performance of query similarity evaluation techniques. We introduce our core contribution, a query regularization techniques in Section 4, and propose a new similarity metric in Section 5. We discuss our experiment results and findings in Section 6. Finally, we conclude by identifying the steps needed to deploy query log summarization into practice using the techniques evaluated in this paper in Section 7.

2. BACKGROUND AND SURVEY

Analyzing query logs mostly relies on the structure of queries [17] 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 [8, 9, 11], performance optimization [12], session identification [13] and workload analysis [14].

Agrawal *et al.* [7] aim to rank the tuples returned by the SQL query based on the context. They create a rule set for the contexts and evaluate the result of queries that belongs to the context according to the ruleset. They consider the similarity of the context and SQL query by creating vectors out of them and measure the cosine distance.

Chatzopoulou *et al.* [11] 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 relational algebra tree as 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, like cosine similarity measure. 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.* [9], on the other hand, build a graph following the database schema from the start to the end tables of joins for each query and group queries based on these graphs using a basic similarity function like Jaccard coefficient. Their aim is again to assist users in writing SQL queries by analyzing query logs. Giacometti *et al.* [8], 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 result columns that correspond to a return result; hence the method depends on having access to the data. Stefanidis *et al.* [10] 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.

Although these methods [7–11] 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,

Table 1: SQL query similarity literature review

Paper title	Motivation	Features	Feature Representation	Distance Function
Context-sensitive Ranking [7]	Query reply importance ranking	Selection Schema Rules	Vector	Cosine similarity
Query Recommendations for OLAP Discovery Driven Analysis [8]	Query recommendation	Difference pairs	Set	Difference query
Recommending Join Queries via Query Log Analysis [9]	Query recommendation	Selection Projection Joins	Graph	Jaccard coefficient on the graph edges
"You may also like" Results in Relational Databases [10]	Recommendation of tuples	Inner product of two queries	Vector	-
The QueRIE System for Personalized Query Recommendations [11]	Query recommendation	Syntactic element frequency	Vector	Jaccard coefficient and cosine similarity
Clustering-based Materialized View Selection in Data Warehouses [12]	View selection optimization	Selection Joins Group By	Vector	Hamming distance
Similarity Measures for OLAP Sessions [13]	Session similarity	Selection Joins Projection Group By	Separate sets of each feature group	Jaccard coefficient
Text Mining Applied to SQL Queries: A Case Study for the SDSS SkyServer [14]	Workload analysis	Term frequency of projection, selection, joins, from, group by and order by	Vector	Cosine similarity

we need to be able to measure pairwise similarity between each query, hence we need a metric that can do so.

Aouiche *et al.* [12] 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.* [13] studies various approaches which define a similarity function to compare OLAP sessions. They focus on comparing session similarity while performing a survey on query similarity metrics, too. They identify selection and join items as the most relevant component 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* and *join* items for queries issued on OLAP datacubes. 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.

Makiyama *et al.* [14] approach query log analysis with a motivation of analyzing the workload on the system, 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 for each query in the dataset. They create a feature vector using the frequency of these terms which they use to calculate the pairwise similarity of queries with cosine similarity emphasizing that euclidean distance has poorer performance. Instead of clustering, they perform the workload analysis with Self-Organizing Maps (SOM).

A summary of these methods is given in Table 1. Our work takes the initiative to evaluate the performance of the three methods [12–14] that offers pairwise similarity metrics in Section 3 due to the lack of direct performance evaluation for the query similarity metrics in the given studies.

3. DCABench

In this section, we introduce DCABench, a new benchmark for query similarity measures and use it to evaluate three query similarity metric from our survey. Our goal is to evaluate how well a query similarity metric captures the intent behind a query. DCABench, the Database Course Assignment Benchmark uses student answers to database course assignments to precisely capture the notion of query intent. Many database courses include homework or exam questions where, students are asked to translate prose into a precise SQL query. The motivation for choosing database course assignments is threefold: First, there is minimal bias or correlation between results, as the prose that students are given has a completely different structure from SQL. Second, ground truth for the intent of each query is clear: students are attempting to accomplish the stated task. Finally, scores (if available) also provide an even more precise metric of how close a query is to the specified task.

In subsection 3.1, we outline the datasets used for DCABench,

as well as our methodology for transcribing student answers from text. Then, in subsection 3.2, 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 intent. Finally, in subsection 3.3, we use DCABench to evaluate the three distance metrics that appear in our literature survey.

3.1 Workloads

Concretely, DCABench relies on two specific query sets: One gathered by IIT Bombay [15] and one gathered at the University at Buffalo and released as part of this paper [1].

The first dataset [15] 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 the University at Buffalo'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 to complete their answers. Unless indicated, we use the full dataset in our experiments. However, since 50% of the grade is the failing criterion, we assume that answers conform with the intent of the question if the grade is over 50%.

A summary of both datasets is given in Tables 2 and 3. Not all student responses are legitimate SQL, and so we ignore queries that can not be successfully parsed by our open-source SQL parser[2]. The number of queries from each dataset that we use is given in the summary tables.

As a final concession for our initial evaluation using DCABench, we ignore constant literals in the queries, replacing them with placeholders. A query with its constant values replaced by a placeholder is called a *query skeleton*. For example, the two queries `SELECT username FROM USER WHERE rank = "admin"` and `SELECT username FROM USER WHERE rank = "moderator"` share the same skeleton because they have exactly the same query structure modulo constant values. Analysis using the set of query skeletons instead of original SQL queries will reduce the number of distinct queries processed and in general will produce similar results. The dataset summary tables indicate how many *distinct* skeletons appear for each query-writing task. One may notice that the number of distinct query skeletons varies in different questions. This is because the level of difficulty of the question given to students as well as the time students spent on answering questions. For easy questions, we expect many virtually identical answers and a small number of skeletons (e.g., as in IIT Bombay question 1). Conversely, the more complex questions asked for the UB Exam dataset admit a much wider range of query variants.

In both of these datasets, the query-writing task is specific. We can expect that student answers to a single question are written with the same intent. 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 clusters to uniformly include answers to the same question.

Question	Total queries	Parsable queries	Skeletons
1	55	54	3
2	57	57	4
3	71	71	56
4	78	78	30
5	72	72	51
6	61	61	5
7	77	65	42
8	79	70	51
9	80	77	67
10	74	74	29
11	69	68	23
12	70	60	14
13	72	70	64
14	67	52	28

Table 2: Summarization of IIT Bombay dataset

Year	Total queries	Parsable queries	Skeletons
2014	117	110	110
2015	60	51	51

Table 3: Summarization of UB exam dataset

3.2 Clustering validation measures

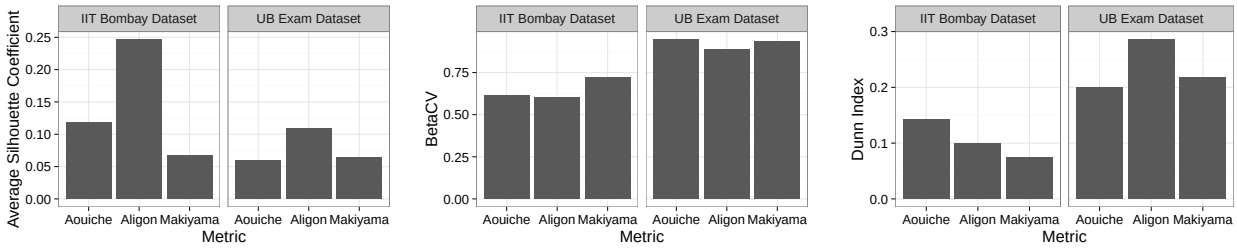
In addition to workload datasets, we define a set of measures to be used for evaluating queries as part of DCABench. Given a set of queries with labels of the their respective tasks and a similarity metric that measures the similarity between two queries, we want to understand how well the metric can (1) put queries that perform the same task close together even if they are written differently, and (2) differentiate queries that perform different tasks.

In order to do that, we use ground truth cluster labels given by the query-writing task that each query belongs to. Using a particular similarity metric, we construct a pairwise distance matrix for the set of queries and subsequently use this matrix to evaluate the similarity metric. With a pairwise distance matrix and a labeled dataset, we can use various clustering validation measures to understand how effective a similarity metric characterizes the partition of a set of queries. Specifically, we will use three clustering validation measures [18, Chapter 17]: Average Silhouette Coefficient, BetaCV and Dunn Index.

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 min-

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

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

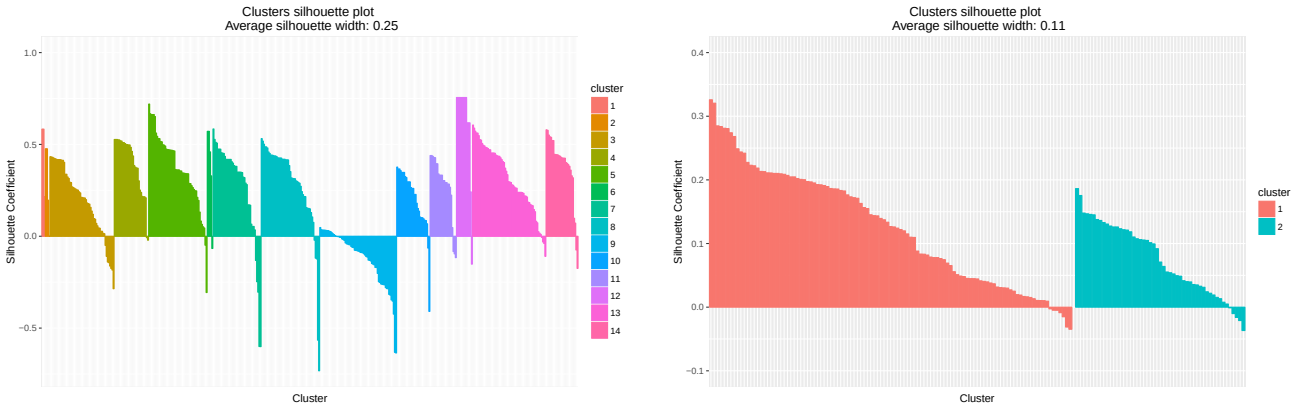


(a) Average Silhouette Coefficient (higher values are better)

(b) BetaCV (lower values are better)

(c) Dunn Index (higher values are better)

Figure 1: Clustering validation measures for each metric



(a) IIT Bombay dataset

(b) UB Exam dataset

Figure 2: Distribution of silhouette coefficients when using Aligon's similarity

imum distance from i to all other data points from other clusters. 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 far away from data points within the same clusters while it is close to the data points from different clusters. Since silhouette's coefficient represents a measure of degree of good match for each data point, to validate the effectiveness of the distance metric given a query partition, we will consider the average silhouette coefficients among all data points (all queries) in the dataset.

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.

3.3 Experimental results

We now apply DCABench to three similarity metrics previously discussed in Section 2: Makiyama's similarity [14], Aligon's similarity [13] and Aouiche's similarity [12]. The aim of the experiment is to evaluate which similarity metric that can capture well the tasks performed by queries. We re-implemented each of these similarity metrics in Java and evaluated them using the three clustering validation measures discussed in subsection 3.2 to evaluate three similarity metrics. In our experiment, the implementation of all three similarity metrics is done in Java. Tests were performed under MacOS 10.11.6 on a 2.2 GHz Intel Core i7 with 16GB of 1600 MHz DDR3 memory using the HotSpot Server JVM 1.8.0_45.

Figure 1 shows a comparison of three similarity metrics using each of the three quality measures (Silhouette, BetaCV and Dunn). As can be seen in Figure 1, Aligon seems to work the best for both datasets under the Average Silhouette Coefficient and BetaCV measures. Aligon also performs well on the Dunn Index, coming in first on the UB dataset, and second-best for IIT Bombay. Especially given that the Dunn Index measures only worst-case performance, Aligon's metric seems to be ideal for the DCABench workloads.

As a reminder, in Aligon's similarity metric, similarity between two SQL queries is computed using a weighted combination of group-by similarity, selection similarity and measure similarity. This shows that even a fairly simple approach can capture well the query semantic. For a closer look of Aligon's similarity metric, Figure 2 shows the distribution of Silhouette coefficients for each query and their respective tasks.

As another observation from the distribution of silhouette coefficients in Figure 2, there are still quite a few queries that

seems to be put into incorrect clusters in both UB Exam dataset and IIT Bombay dataset, especially for question 7 of IIT Bombay dataset. This leads us to the next question we address in this paper: Can we improve these numbers? In the next section, we present a query regularization step which improves the effectiveness of these three similarity metrics at measuring similarity of intent.

4. REGULARIZATION

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, the same end goal can be expressed through many different equivalent queries. Consider the following (contrived) example:

EXAMPLE 1. *Sub-query nesting*

1. `SELECT R.r WHERE R.r>1 FROM (SELECT * FROM R WHERE R.r<4);`

2. `SELECT R.r WHERE R.r<4 FROM (SELECT * FROM R WHERE R.r>1);`

In spite of being guaranteed to produce identical results, the ASTs for both queries are quite different with respect to sub-trees they contain. Let's consider another slightly more subtle example:

EXAMPLE 2. *OR-UNION Ambiguity*

1. `SELECT * FROM R WHERE R.r<1 OR R.r>4;`

2. `SELECT * FROM R WHERE R.r<1 UNION SELECT * FROM R WHERE R.r>4;`

For this example, the two queries produce identical results because `R.r < 1` and `R.r > 4` are mutually exclusive conditions.

Although general query equivalence is NP-complete [19], we can still significantly improve clustering quality by standardizing certain SQL features into "more regular" forms. We note that this regularization process is orthogonal to the choice of distance metric and our experiments show improvements for most of the distance metrics we compare against. In this section, we describe the transformations that we apply to regularize queries and address some of the limitations of regularization.

4.1 Regularization Rules

Standardize Expressions. 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 will exploit later.

Second, we attempt to reduce expressions. For example, the expression 1+1 would be replaced by 2. We also attempt to detect tautologies and contradictions in boolean-valued expressions, allowing us to replace the entire expression with a fixed value.

Similarly, expressions involving commutative and associative operators (e.g., $+$, $\times$, $\wedge$, and $\vee$) have many equivalences. For example, there are there are 6 ways to write the expression $A + B + C$. We standardize such expressions by using the hash of each operand to define a canonical order.

Finally, we remove redundant expression elements: (1) `BETWEEN` predicates are replaced by the corresponding comparisons. (2) Greater-than operators ($>$, $\geq$) are replaced with their corresponding lesser-than operators ($<$, $\leq$). (3) common functions like `isnull` are inlined into the expression. For example `isnull(X,Y)` becomes `CASE WHEN X IS NULL THEN Y ELSE X END`.

Flatten FROM-Nesting. As in example 1, nested sub-queries can exist in a `FROM` clause. When the sub-query is not an aggregate or `DISTINCT` query, it can be flattened into its parent expression. Terms in the sub-query's `SELECT` clause (i.e., target terms) are inlined into the parent query, tables in the sub-query's `FROM` clause are added to the parent query, and the sub-query's `WHERE` clause is combined conjunctively with the parent.

Flatten Predicate-Nesting. Nesting can also occur in expressions, for example through the `EXISTS` predicate. Consider the following two equivalent queries:

EXAMPLE 3. *EXISTS-JOIN ambiguity*

1. `SELECT DISTINCT * FROM Student WHERE EXISTS(SELECT * FROM Score WHERE Score.grade>60 AND Score.StudentID=Student.ID)`

2. `SELECT DISTINCT * FROM Student JOIN Score WHERE Student.ID=Score.StudentID AND Score.grade>60`

The equivalence between these two queries is an example of nested query de-correlation [20]. We apply a similar form of nested query de-correlation, albeit in two independent stages. Because not all nested queries can be de-correlated safely, the first stage serves to eliminate redundancy. The `EXISTS` predicate is general enough to capture the remaining three nested query predicates (`IN`, `ANY`, and `ALL`). Queries including the latter three 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)`

Flatten Sub-queries in `EXISTS`. Next, if possible, `EXISTS` predicates are de-correlated. To ensure that we can safely de-correlate the predicate, we check whether two safety constraints are satisfied:

1. `EXISTS` predicates must be in a purely conjunctive `WHERE` clause. No disjunction or nested negation.

2. The parent query is either a `SELECT DISTINCT` or a duplicate-insensitive aggregate [21] (e.g. $max(1,1) = max(1)$, but $sum(1,1) \neq sum(1)$)

The decorrelation process itself moves the query nested in the `EXISTS` into the `FROM` clause of its parent query. The (formerly) nested query's `WHERE` clause is then moved into the parent's `WHERE` clause. 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
```

Flatten Sub-queries in `NOT EXISTS`. Queries with negated `EXISTS` predicates (i.e., `NOT EXISTS`), can also be de-correlated if the same set of safety constraints is satisfied. To de-correlate a `NOT EXISTS` predicate, we use the set-difference operator `EXCEPT`. If the input is of the form:

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

then the output will be of the form

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

OR-UNION transform.

Recall the equivalence between logical OR and UNION in Example 2. Naively, we might convert the DNF-form predicates into UNION queries:

TRANSFORMATION 1. $\sigma_{(C_1 \vee \ldots \vee C_N)}(Q) \rightarrow \bigcup_{i \in (1 \ldots N)} (\sigma_{C_i}(Q))$

However, duplicates caused by the possible correlation between clauses in DNF will break the equivalence of this rewrite. Consider the following query

EXAMPLE 4. *Correlated clauses*

```
SELECT sum(score) FROM exam WHERE score>61 OR pass=1
```

Suppose criterion for passing the exam is 60, then students who pass the exam are highly overlapped with students with score higher than 61 and the rewritten query is not equivalent. As a result, we should only convert mutually exclusive ORs.

Naively, we might generate a sequence of mutually exclusive OR statements using Shannon expansion [22]. Unfortunately, Shanon expansion can create an exponential number of clauses, further exploding the number of features that need to be created.

A second alternative is to define a new form of duplicate-sensitive union, essentially a form of set-union (i.e., `UNION DISTINCT`) that relies on tuple provenance [23] to limit duplicate values. Although it may not be reasonable to expect queries with this duplicate-sensitive union to be evaluated efficiently, the query itself is never evaluated, but simply compared against other queries.

It turns out that in cases without sub-queries nested in predicates, there is still value in remapping `OR`s into `UNION`s. Consider the following instance of *tautology-based SQL injection* [24].

EXAMPLE 5. *Tautology in Disjunction*

1. `SELECT * FROM R WHERE R.r>1 OR 1=1`

2. `SELECT * FROM R WHERE R.r>1`

Although the above two queries are quite structurally similar, the set of rows returned will be very different. Transforming OR into UNION in example 5 will partition the query into two independent parts, making it easier to detect that one of the partitions returns all rows of R.

Aggregate functions will prevent us from fully serving this purpose. Adding aggregate *max* to query 1 in example 5.

EXAMPLE 6. *OR-UNION transform under aggregate*

```
SELECT max(R.r) FROM R WHERE R.r>1 OR 1=1
SELECT max(R.r) FROM ( (SELECT * FROM R WHERE R.r>1)
UNION (SELECT * FROM R WHERE 1=1) )
```

UNION stays in FROM clause after regularization because of *max*. To pull up UNION above *max(R.r)*, we make copies of *max(R.r)* and distribute them into each component query under UNION. Note that this transformation does not retain equivalence, we need one more step of *max* aggregate to recover the correct result. Trade-off is that pulling up UNION to the top is friendlier for analysis and comparison purpose. In short, our goal of regularization is to turn any query into a UNION of *conjunctive queries* [25] without any sub-query nesting.

Next we discuss regularization for CASE expressions as they add in ambiguity.

Contingent values and CASE expression translation. `CASE` expressions introduce if-then-else logic flow control in SQL language. Consider the two `CASE` expressions

EXAMPLE 7. *`CASE` expression ambiguity*

1. *`CASE WHEN` $A.a < 3$ `THEN` X `WHEN` $A.a < 4$ `THEN` Y `ELSE` Z*

2. *`CASE WHEN` $A.a \geq 4$ `THEN` Z `WHEN` $A.a \geq 3$ `THEN` Y `ELSE` X*

In `CASE` statements, conditions are implicitly correlated because a condition is evaluated only if all of its predecessors are not satisfied. Rephrasing the `CASE` as an unordered set of disjoint conditions helps to resolve ambiguity from this implicit correlation by making the correlation explicit. Concretely, each condition is conjuncted with the negation of its predecessors in the statement. Continuing the example above, in statement 1, the second `WHEN` clause becomes $\neg(A.a < 3) \wedge A.a < 4$.

As a result of transformation, the `WHEN` clauses of both statements mirror one-another, checking for values in the range $[3, 4]$. Also note that the resulting `CASE` statement partitions its input set, and can easily be re-cast as a `UNION` (also a common technique in query optimization). Given a query of the form

```
SELECT ... CASE WHEN X1 THEN A1 ...
                WHEN XN THEN AN ...
```

we emit a query of the form

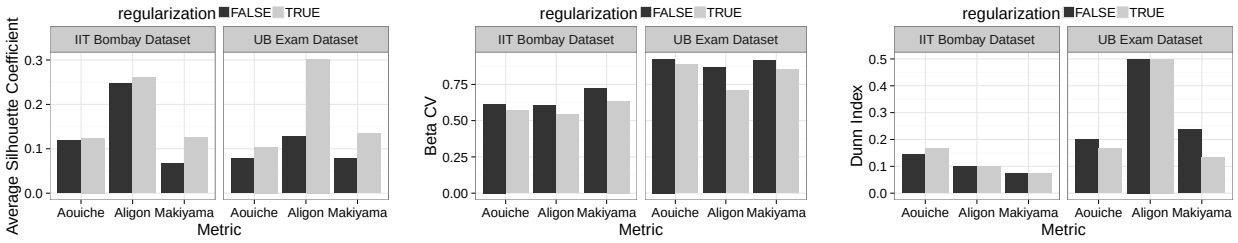
```
(SELECT A1 ... WHERE X1) UNION ... UNION
   (SELECT AN ... WHERE (NOT X1) AND ... AND XN)
```

In the above transformation, Boolean-valued `CASE` statement (i.e., those already appearing in a where clause) can be flattened as a form of implication: $A \rightarrow B \equiv (\neg A \vee B)$.

4.2 Evaluation

We next evaluate the effectiveness of regularization by applying it to each of the three metrics described in Section 3. We use DCABench, and compare the quality of each measure both with and without regularization. Note that questions in UB exam require student to write queries fulfilling textually similar but semantically distant intents in order to test whether students comprehend the subject. Hence we expect regularization helps all these three algorithms correctly distinguishing answers written under different questions. However, as student answers are noisy and some do not conform with the intent required, we minimize noise in the clustering results by considering only student answers in the UB exam dataset that received a grade of over 50%.

Figure 3 shows the three validation measures for each of the three similarity metrics, both with and without regular-



(a) Average Silhouette Coefficient (the larger value is better)

(b) BetaCV (the smaller value is better)

(c) Dunn Index (the larger value is better)

Figure 3: Clustering validation measures for each metric with and without regularization step

ization. As we observe from the Figure, regularization significantly improves the Average Silhouette Coefficient and BetaCV measures for all similarity metrics. The Dunn index is relatively unchanged for the IIT Bombay dataset, and shows slight signs of worsening with regularization on the UB Exam dataset. This is not unexpected: Regularization simplifies queries and as a result all queries become more similar.

Note that for (a) in Figure 3 on UB exam dataset, the boost in performance of the other two algorithms is not comparable to Aligon. Because only Aligon treats features created under different functional clause of the query(e.g. `SELECT`, `FROM`) as independent and not comparable. In UB dataset, nested sub-queries are frequent and regularization flattens sub-queries and merges their functional clauses together with the parent. Aligon essentially generates a separate feature vector for each merged functional clause. For the other two algorithms, total weight of features from vector of `SELECT` clause can be overwhelmed by the vector of `FROM` clause when vectors are merged into one. Generally speaking, merged `SELECT` clause offers larger distinguishing power than merged `FROM` clause after regularization. Hence the benefit from regularization is degraded for the other two algorithms.

5. SIMILARITY METRIC

As a declarative language, the abstract syntax tree (AST) of a SQL statement acts as a proxy for the intent of the query author. We thus argue that structural similarity is 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 intent has seen substantial use already, particularly in the translation of natural language queries into SQL [26].

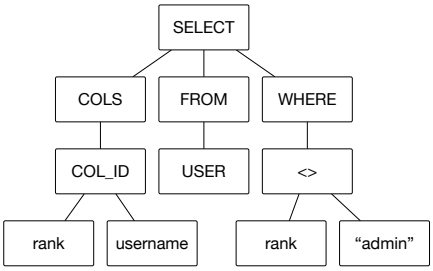


Figure 4: An abstract syntax tree of query `SELECT rank, username FROM USER WHERE rank <> "admin"`.

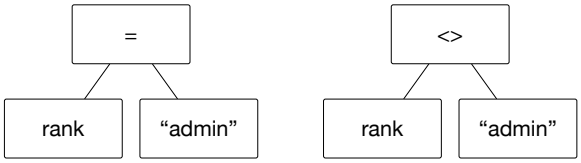
As explained in Section 2, existing similarity metrics mainly utilize the columns from specific parts of the query; making use of the combination of projection, selection, joins, or group by items. However, we claim that having the same set of items does not necessarily imply that these queries have similar intents, even when they operate on similar domains. Consider the following example:

Example 8. *Queries accessing same columns*

1. `SELECT username FROM USER`
   `WHERE rank = "admin"`

2. `SELECT COUNT(username), rank FROM USER`
   `WHERE rank <> "admin"`
   `GROUP BY rank`

For the remainder of this paper, we will use Q to denote both the query itself as well as its tree encoding. The queries Q_1 and Q_2 in Example 8 share the same set of columns; `username`, and `rank`. Nevertheless, it is obvious that Q_1 aims to list all usernames with admin rank, while Q_2 intends to count the number of users for every rank except the ones with admin rank. While having overlapping sets of items indicates a potential similarity, the intent behind these queries are completely different. This problem can easily be solved if we utilize the query ASTs. The AST of `WHERE` clause expressions of both queries can be seen on Figure 5.



(a) The subtree of expression Q_1:`rank = "admin"`

(b) The subtree of expression Q_2:`rank <> "admin"`

Figure 5: Two example subtrees

According to other similarity metrics surveyed in Section 2, the expressions `rank = "admin"` and `rank <> "admin"` are the same, because the only grammar item they would consider while calculating pairwise similarity is the column `rank`. We, on the other hand, argue that we need to consider all items and all expressions. Table 4 shows all features we can utilize.

When we construct feature vectors v_1 and v_2 by counting the appearance of each feature in Q_1 and Q_2 respectively, we end up with the vectors listed in Table 5.

Table 4: Feature set of Q_1 and Q_2

Feature	Item
1	*rank*
2	=
3	<>
4	*"admin"*
5	*rank = "admin"*
6	*rank <> "admin"*

Vectors	Features					
	1	**2**	**3**	**4**	**5**	**6**
v_1	1	1	0	1	1	0
v_2	1	0	1	1	0	1

Table 5: Feature vectors constructed for queries Q_1 and Q_2 (See Example 8).

Pairwise similarity between the two queries can now be computed using cosine similarity:

$$sim(Q_1, Q_2) = cos(\boldsymbol{v}_1, \boldsymbol{v}_2) = \frac{\boldsymbol{v}_1 \cdot \boldsymbol{v}_2}{||\boldsymbol{v}_1|| \cdot ||\boldsymbol{v}_2||} = 0.5$$

This construction process is essentially the feature extraction scheme based on the Weisfeiler-Lehman test of isomorphism on graphs.

5.1 Weisfeiler-Lehman algorithm

An ideal distance measure takes into account the level of similarity or overlap between these tree encodings and their substructures. 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 [16] illustrated as Algorithm 1. Instead of comparing all possible subgraphs of Q_1 against all possible subgraphs of Q_2, the WL algorithm restricts itself to specific types of subgraphs. It takes the set of all query tree encodings T as an input and it outputs a set of labels for each tree encoding where all the identical subgraphs in the query log are labeled with the same value.

Given a query Q, let $N \in Q$ denote a node in Q. N is initially labeled with the SQL grammar symbol that N represents. The *i-descendent* tree of N: $desc(N, i)$ is the sub-tree rooted at N, including all descendants of N in Q up to and including a depth of i. The features of *i-descendent* sub-tree rooted at N: $feature(N, i)$ is all of the possible i-descendent trees that can be generated from N.

EXAMPLE 9. *Given the tree in Figure 4, desc(*`COL_ID`*, 1) is the tree containing the nodes* `COL_ID`*,* `rank`*, and* `username`*. feature(N, i) is the expressions (*`COL_ID`*), (*`rank`*), (*`username`*) and (*`rank, COL_ID, username`*).*

EXAMPLE 10. *Given the tree in Figure 4, desc(*`WHERE`*, 2) is the tree containing the nodes* `WHERE`*,* `<>`*,* `rank`*, and* `"admin"`*. feature(N, i) is the expressions (*`WHERE`*), (*`<>`*), (*`rank`*), (*`"admin"`*), (*`WHERE, <>`*), (*`rank, <>, "admin"`*) and (*`WHERE, rank, <>, "admin"`*).*

The WL algorithm identifies a query Q by all possible i-descendent trees that can be generated from Q:

$$id(Q) = \{\ desc(N, i) \mid N \in Q \wedge i \in [0, depth(Q)]\ \}$$

Algorithm 1 Weisfeiler–Lehman Algorithm

```
1: procedure WEISFEILER–LEHMAN
2:     for each tree Q ∈ T do
3:         H ← depth(Q)
4:         for h ← 1; h ≤ H; h++ do
5:             for each node N ∈ Q do
6:                 f ← CreateFeature(
                       expressions in {N ⋃ ∀ N.children})
7:                 if featureSet.get(f) ≠ null then
8:                     N ← featureSet.get(f)
9:                 else
10:                    featureSet.put(f)
11:                end if
12:                if IsLeaf(∀N.children) then
13:                    N.IsLeaf = true
14:                end if
15:            end for
16:        end for
17:    end for
18: end procedure
```

Here $depth(Q)$ is the maximum distance from the root of Q to a leaf. To make this comparison efficient, subtrees are deterministically assigned a unique integer identifier, and the query is described by the bag of Q's i-descendent tree identifiers. Thus two query trees with an isomorphic subtree will both include the same identifier in their description. The bag of identifiers is encoded as a feature vector and allows a distance function like euclidean distance or cosine distance to measure the similarity (or rather dis-similarity) of two queries.

The algorithm operates from top to the bottom, exhaustively identifying what expressions each node should carry. The leaf (bottom) nodes are always the grammar atoms, hence cannot be splited into smaller items. Once a full iteration from top to the bottom of the AST finishes, each node would carry its own value and the expression derived from its children's. Once the children of a node finalize, namely, they don't get assigned any new expressions, that parent node is marked as finalized, too. Until all of the nodes are finalized in the AST, the algorithm goes on. Hence, from the smallest grammer atom to the full query itself become a feature. This process guarantees the extraction of all possible expressions in the query as a feature deterministically.

Figure 6 shows how the WL algorithm is applied on the AST of the query given in Figure 4. First, every distinct node of the AST gets labeled with a unique integer. If the same node appears more than once, each instance is labeled with the same integer. As the algorithm progresses, the dotted box emphasizes the region being examined, while the text below represents new labels being synthesized from existing labels. Grey nodes have been fully labeled, while white nodes are still being processed.

5.2 Clustering using the similarity metric

In our experiments, we use cosine similarity as the similarity metric. We could have used Euclidean or Manhattan distance, too, but in that case we would need to have an additional normalization step; for instance, when we compare two queries with only one feature different from each other, our result would be 2 units. However, when we compare two queries with hundreds of features in common, but

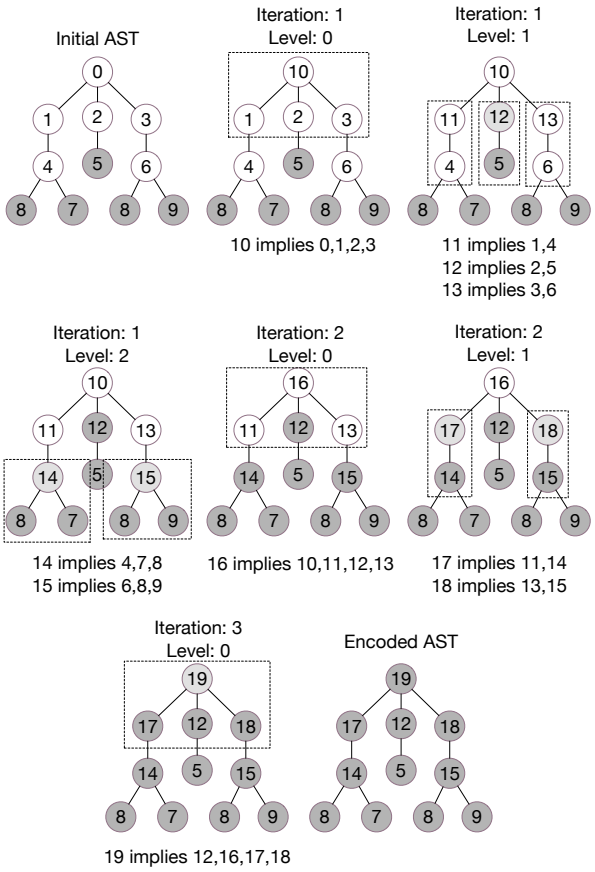


Figure 6: Weisfeiler-Lehman algorithm applied on AST given in Figure 4.

only three features different, our result would be 3 units, hence we would classify the latter pair more different than the former pair. We can perform clustering of similar queries with the pairwise similarity matrix of a query set by creating the feature vectors with WL algorithm, and then calculating pairwise distances with cosine similarity.

The accuracy of this method to extract important features can be increased in various ways. In Section 4, we talk about how we can exploit the flexibility of SQL queries by regularization. We also discuss what can be done to improve the algorithm's performance in Section 6. In the following section, we show how this similarity metric compares against the other similarity metrics evaluated in this paper.

5.3 Evaluation

In order to evaluate our the effectiveness of WL algorithm in characterize query similarity, we compare it against three other metrics which have been mentioned in Section 3. Figure 7 shows the comparison of all four metrics using three different clustering validation measures. We, again, use the UB Exam dataset with the queries graded over 50% and IIT Bombay dataset for the same reasons explained in Section 4.2. As observed in the figure, even though WL algorithm is not the best across three different measures, it is quite competitive in comparison with other similarity metrics. This shows WL algorithm has a great potential for improvement when it is only used in its straightforward form and hasn't been tweaked much to adapt with the query data.

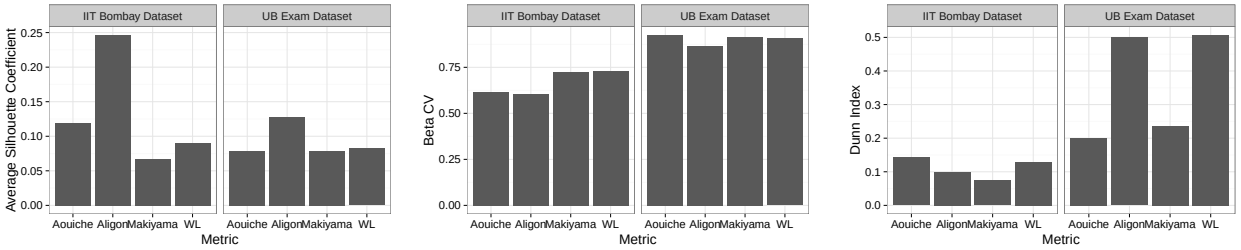
Beside it, we also want to explore how well WL algorithm works when using regularization step. For this reason, we perform another experiment to compare the effectiveness of WL algorithm with and without regularization step. Figure 8 shows the result of this experiment. As observed in the figure, adding regularization step can yield an improvement in performance of WL algorithm when considering Average Silhouette Coefficient or BetaCV as a quality measure. On the other hand, there is no clear improvement or even a reduction in clustering quality when Dunn Index is used as a clustering validation measure. However, the same argument from subsection 4.2 can be applied here when noting that Dunn Index only measures the extreme cases in the dataset while Average Silhouette Coefficient and BetaCV consider all queries in dataset when computing the measures. Therefore, we may conclude that even though some queries at the borderline of cluster may suffer a decrease in clustering quality, adding regularization step into WL algorithm will increase in its overall capability of capturing query semantics and put queries that perform the same task spatially close together.

For understanding the computational time to process a set of queries and producing pair-wise distance matrix for WL algorithm and its relative processing time to other similarity metrics, we measure and report the running time of each similarity metric with and without regularization. For all metrics, the running time is measured 5 times and the average value is reported. Figure 9 shows the comparison in terms of running time among different similarity metrics. As observed in the figure, when using regularization, the running time is consistently reduced except for the case of Makiyama's similarity metric when IIT Bombay dataset is used. For UB Exam dataset, probably because of the small number of queries, the running time is almost comparable among four similarity metrics. With IIT Bombay dataset where the number of queries is nearly three times more than the number of queries in UB Exam dataset, the difference in running among different similarity metrics are more notable. Although WL algorithm is slowest among four metrics when regularization step doesn't apply, its running time decreases drastically when using regularization. This can be explained by noting that the regularization step allows the query to be converted into a more standardized form which subsequently is more efficient in extracting features and leads to a reduction in running time.

6. DISCUSSION

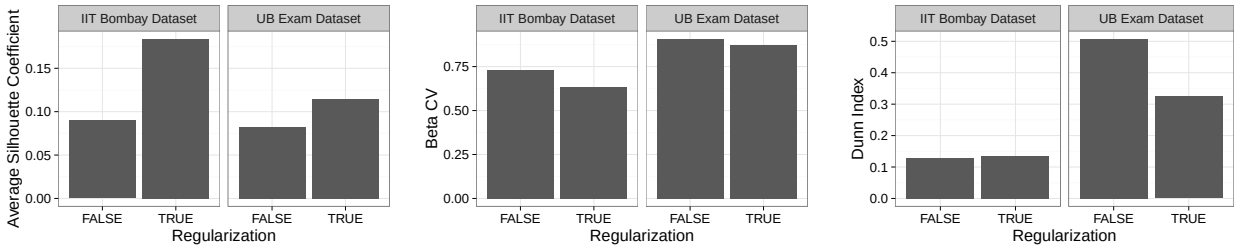
We have surveyed several similarity metrics for clustering queries and focused on three methods that offer an end-to-end similarity metric. The survey shows that most of the metrics make use of selection and join predicates and consider them as the most important items for similarity calculation. Group-by predicates follows them closely while projection items take the third most important item set. There are other possible feature sets that can be used in queries like the tables accessed or the AST of a query, but these feature sets are generally overlooked.

As shown in Figure 1, apart from the Dunn index value for IIT Bombay dataset, Aligon *et al.* [13] performs the best amongst the surveyed methods. The success of this method can be attributed to the choice of the feature set: selection, joins, group by and projection items separately have



(a) Average Silhouette Coefficient (the larger value is better) (b) BetaCV (the smaller value is better) (c) Dunn Index (the larger value is better)

Figure 7: Comparing the performance of WL algorithm with other similarity metrics



(a) Average Silhouette Coefficient (the larger value is better) (b) BetaCV (the smaller value is better) (c) Dunn Index (the larger value is better)

Figure 8: Comparing performance of WL algorithm with and without regularization step

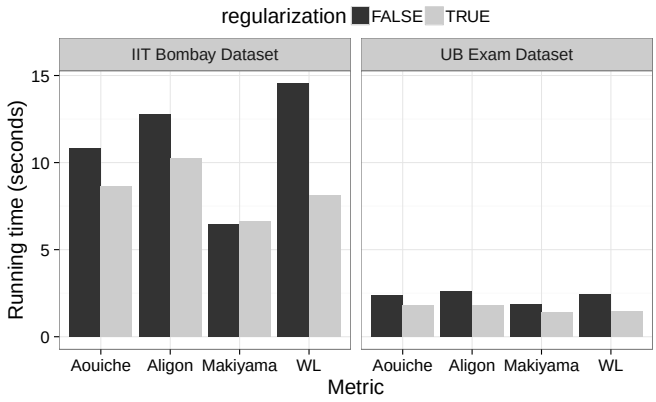


Figure 9: Running time comparison among different query similarity metrics with and without regularization

their own weightings in the general calculation of the similarity. Although Aouiche *et al.* [12] have a similar strategy; making 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.* [14], on the other hand, follows Aligon *et al.* [13] 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.

As can be seen in Tables 2 and 3, as the complexity or difficulty of the question increases, the number of query skeletons also increases, i.e., students find different ways to solve the same problem. Especially, in Table 3, 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 increase. Figure 3 shows that our assumption that regularizing queries will improve clustering quality is correct. Our proposed regularization algorithm improves the clustering quality of all three metrics. One caveat is that regularization is only suitable for query comparison purpose as some of its transformations do not retain equivalence. In addition, once a nested sub-query is flattened and merged with its parent, aliases in the parent query pointing to it become invalid. At the same time, names across different query bodies are merged and name uniqueness is not guaranteed any more. Skewed names in the query will make query comparison error-prone. Hence we apply a recommended step before regularization that renames all entities in the query containing aliases.

In Section 5, we introduce a new similarity metric that is based on the feature extraction scheme based on the Weisfeiler–Lehman test of isomorphism on graphs where we make use of query ASTs instead of atomic features. Even with this naive approach, the performance of the metric is competitive with the metrics surveyed in this paper as can be seen in Figure 7. The quality is further improved with the regularization scheme we proposed as shown in Figure 8. Furthermore, the algorithm can be benefit greatly from feature weighting (e.g. appointing higher weights to the features created from selection and joins) and dimensionality reduction with PCA, ICA or other feature selection techniques. Just like Makiyama *et al.* [14], we create many unimportant features along with the important ones, and feature selection can improve the accuracy of the metric greatly.

7. CONCLUSION AND FUTURE WORK

The focus of this paper is to summarize large query logs by clustering queries by similarity of intent. We describe a benchmark, DCABench, that captures the notion of query intent using student answers to query-construction problems, and use this benchmark to evaluate three query similarity metrics. We also propose a regularization technique for standardizing query representations. Through further experiments with DCABench, we show that regularization significantly improves all three query similarity metrics, and in particular the Aligon metric.

We identify the shortcomings of the current similarity metrics and introduce a new technique based on Weisfeiler–Lehman approximate graph isomorphism algorithm to create feature vectors. We clearly demonstrate that the resulting query metric can be effective at clustering queries with similar intent together. We show that its base performance is comparable to the other methods, and that it scales well.

The approaches described in this paper only represent the first steps towards tools for summarizing logs by intent. Concretely, we plan to extend our work in several directions: First, we will explore new feature weighting strategies and new labeling rules in order to capture the intent behind logged queries better. Second, we will examine user interfaces that better present clusters of queries — Different feature sorting strategies in Frequent Pattern Trees (FP Trees) [27] in order to help the user distinguish important and irrelevant features, for example. Third, further exploration on various kinds of statistics captured in FP Trees will help us in determining the quality of the cluster, weighting of features, and visualizing the summary of the clusters. Lastly, we will investigate the effect of temporality 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.

8. REFERENCES

[1] Nicolas Bruno and Surajit Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *ACM SIGMOD*, 2005.

[2] Oliver Kennedy, Jerry Ajay, Geoffrey Challen, and Lukasz Ziarek. Pocket Data: The need for TPC-MOBILE. In *TPC-TC*, 2015.

[3] Gokhan Kul, Duc Luong, Ting Xie, Patrick Coonan, Varun Chandola, Oliver Kennedy, and Shambhu Upadhyaya. Ettu: Analyzing query intents in corporate databases. In *WWW Companion*, 2016.

[4] Cynthia Dwork. *ICALP 2006, Proceedings, Part II*, chapter Differential Privacy. Springer, 2006.

[5] Wolfgang Gatterbauer. Databases will visualize queries too. *pVLDB*, 2011.

[6] Qingsong Yao, Aijun An, and Xiangji Huang. Finding and analyzing database user sessions. In *DASFAA*, 2005.

[7] Rakesh Agrawal, Ralf Rantzau, and Evimaria Terzi. Context-sensitive ranking. In *ACM SIGMOD*, 2006.

[8] Arnaud Giacometti, Patrick Marcel, Elsa Negre, and Arnaud Soulet. Query recommendations for OLAP discovery driven analysis. In *ACM DOLAP*, 2009.

[9] X. Yang, C. M. Procopiuc, and D. Srivastava. Recommending join queries via query log analysis. In *IEEE ICDE*, 2009.

[10] Kostas Stefanidis, Marina Drosou, and Evaggelia Pitoura. "You May Also Like" results in relational databases. In *ACM DOLAP*, 2009.

[11] Gloria Chatzopoulou, Magdalini Eirinaki, Suju Koshy, Sarika Mittal, Neoklis Polyzotis, and Jothi Swarubini Vindhiya Varman. The QueRIE system for personalized query recommendations. *IEEE Data Eng. Bull.*, 2011.

[12] Kamel Aouiche, Pierre-Emmanuel Jouve, and Jérôme Darmont. Clustering-based materialized view selection in data warehouses. In *ADBIS*, 2006.

[13] Julien Aligon, Matteo Golfarelli, Patrick Marcel, Stefano Rizzi, and Elisa Turricchia. Similarity measures for OLAP sessions. *Knowledge and information systems*, 2014.

[14] Vitor Hirota Makiyama, M Jordan Raddick, and Rafael DC Santos. Text mining applied to SQL queries: A case study for the SDSS SkyServer. In *SIMBig*, 2015.

[15] Bikash Chandra, Bhupesh Chawda, Biplab Kar, K. V. Reddy, Shetal Shah, and S. Sudarshan. Data generation for testing and grading SQL queries. *VLDBj*, 2015.

[16] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *JMLR*, 2011.

[17] Ashish Kamra, Evimaria Terzi, and Elisa Bertino. Detecting anomalous access patterns in relational databases. *VLDBJ*, 2007.

[18] Mohammed J Zaki and Wagner Meira Jr. *Data mining and analysis: fundamental concepts and algorithms*. Cambridge University Press, 2014.

[19] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[20] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *IEEE ICDE*, 1996.

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

[22] Claude. E. Shannon. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 1949.

[23] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *ACM PODS*, 2007.

[24] William G Halfond, Jeremy Viegas, and Alessandro Orso. A classification of sql-injection attacks and countermeasures. In *IEEE ISSSE*, 2006.

[25] Ashok K. Chandra and Philip M Merlin. Optimal implementation of conjunctive queries in relational data bases. In *ACM STOC*, 1977.

[26] Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *pVLDB*, 2014.

[27] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *DMKD*, 2004.