Efficient Query Recommendation

Gregory Borodin
Efficient Query Recommendation

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Gregory Borodin

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tevet 5772 Haifa January 2012
This research was carried out under the supervision of Prof. Yaron Kanza, in the Faculty of Computer Science.

# Contents

Abstract ................................. 1

Abbreviations and Notations .......... 3

1 Introduction ......................... 5

2 Related Work .......................... 11
   2.1 Session-Based Recommendations ... 11
   2.2 Query Clustering .................. 12

3 Framework ............................. 15
   3.1 Query Search ...................... 15
      3.1.1 Syntactic Search .......... 15
      3.1.2 Semantic Search ........ 16
      3.1.3 Semantic Similarity versus
           Syntactic Similarity ..... 17
   3.2 Additional Definitions and Notations ... 17

4 Methods Description .................. 21
   4.1 Tree Edit Distance ............ 21
      4.1.1 Edit Operations and
           Tree Edit Distance .... 21
      4.1.2 Syntax Trees are not
           Ordered Trees .......... 24
      4.1.3 Original TED and
           Heuristics Search
           Algorithms .......... 26
      4.1.4 Complexity Analysis .... 36
   4.2 Attribute Analysis .............. 38
      4.2.1 Advantages and
           Drawbacks of the
           Attribute Analysis
           Method ........... 40
      4.2.2 Complexity .............. 40
   4.3 Vector Model .................... 41
      4.3.1 An Improved Vector Model .. 42
      4.3.2 Using Full Attribute Names .. 43
      4.3.3 Discussion .............. 44
   4.4 Results Analysis ............... 44
      4.4.1 Similarity Based on a
           Single Database Sample .. 45
      4.4.2 Database Samples
           Generation ............. 46
Abstract

In large organizations, frequently the database has a large and intricate schema, hence, formulating queries is cumbersome. In such organizations, users can benefit from finding relevant queries in the query log of the database—queries that can serve as an initial example for query formulation, or queries that were written by experts and elucidate how to write the query in an optimized manner. In this work we describe a recommendation system that assists users by providing a query search capability over a repository of SQL queries. We distinguish between two types of search—semantic search and syntactic search. In semantic search, queries are deemed similar if they were written for the same task. In syntactic search, queries are considered similar if their formulation looks syntactically similar, to an ordinary user. In this work we illustrate these two types of similarity and the differences between them. We developed accurate methods for testing each type of similarity between given queries, and two additional less accurate but more efficient methods, which we use to improve the performance of the search. We present these methods and we provide an experimental evaluation that demonstrates the effectiveness of our methods. Finally, we demonstrate a recommendation system that assists users finding queries that are similar to a given user query, under each one of the two search semantics. In addition, the recommendation system allows searching for queries from scratch, where the search specifications are the syntactic fragments of the queries or the results of the queries that they yield on generated database samples.
Abbreviations and Notations

\( \text{sim}_{TED}(q_1, q_2) \) — similarity between queries measured using the Tree Edit Distance method.

\( \text{OTED}(t_1, t_2) \) — Ordered Tree Edit Distance between the syntax trees \( t_1 \) and \( t_2 \).

\text{unordered node} — a node whose children can be reordered without changing its semantics.

\( \text{TED}(t_1, t_2) \) — Tree Edit Distance of \( t_1 \) and \( t_2 \).

\( \text{sim}_{VM}(q_1, q_2) \) — similarity between queries measured using the Vector Model method.

\( \text{sim}_{AA}(q_1, q_2) \) — similarity measured using the Attribute Analysis method.

\( P_q \) — is the set of all the attributes that appear explicitly or implicitly in the Select clause of the query \( q \).

\( C_q \) — is the set of all the attributes that appear in the constraints of the query \( q \).

\( \text{sim}_{RA}(q_1, q_2) \) — similarity between queries measured using the Result Analysis method.

\( r^D_q \) — the result of a query \( q \) over a database sample \( D \).

\( |r^D| \) — the number of rows of a result \( r^D \).
Chapter 1

Introduction

Modern database management systems provide advanced mechanisms to assist users in organizing, storing, managing, and retrieving data. They provide, however, only limited capabilities for managing the queries that users issue on the data [16]. Queries are usually composed and debugged by experts who are familiar with the database, and in many cases queries are repeatedly issued by applications. However, there are environments where users of large databases (scientific data centers [15, 19, 21], massive data logs, etc.) compose and execute ad hoc queries on large volumes of data. Frequently, in such cases, users are not familiar with the schema of the database that they query, due to its complexity, and consequently they should leverage the knowledge about queries that others have previously executed on the data. A database management system typically stores a log containing the queries that were previously posed in the system. It, thus, can be beneficial for users to find queries that are helpful for them in the log. We refer to queries that are stored in the system, in the log or in some other repository, as log queries or repository queries, in contrast to user queries, which are the queries initially posed by the user before leveraging the queries in the system.

To help a user, we propose an approach where the system finds repository queries that are similar, in some sense that will be defined later, to the user query. Such approach can be useful in the following cases. When queries are being stored in the log, searching for specific queries in the log can help tracing activities in which queries are involved. For instance, when a company manager wants to examine if queries of some type were posed in the system recently, e.g., to verify that workers do not misuse the company data. In a medical care center, the manager can search for queries that retrieve medical data of celebrities, to examine that only people who really need to access the data indeed access it. In a bank, the manager can look for queries about inactive accounts, to try and detect frauds or misuse of inactive accounts by his employees. In such cases, the detection can be applied in real time, to trigger an alert in case of a suspicion of a breach.

Finding similar queries can also be useful when a user detects an error in some query. If this happens, it would be beneficial to find if similar erroneous queries were posed to
the system, and when they were posed, to try and reduce the damage caused by false
answers. Finding similar queries also helps to detect cases where students of a database
course copied the answer to an homework assignment from another student.

Another use for detecting similarity between queries is to assist users in query
formulation. For example, consider a user who wants to retrieve some data from a
database with a complicated schema, and suppose that the user is not familiar with
the schema. The user may start by writing a simple, and not necessarily correct, query.
Then, if the system automatically finds in the log queries similar to the user query and
provides them as a recommendation, the user can either use one of the existing queries,
as is, or leverage the discovered similar queries to write her own query.

As an alternative to a search that starts by composing a complete query, the user
may quickly write or select pieces of a query and use these to find log queries that
contain those pieces. Then, the user can modify the discovered queries, and by this
speed up query composition. Furthermore, after the user has written a query, the system
may automatically recommend log queries that retrieve almost the same data, or that
in some sense, perform a similar task, as the user query. Then, the user may decide
whether to use a recommended query instead of the initial query, since the recommended
query has been written by an expert and consequently is expected to be more efficient
than the original user query.

In these examples, users benefit from searching for log queries that are similar to
the queries they compose. However, in different cases, there is a need for different
type of similarity between queries. In this work, we define two types of similarities
between SQL queries: syntactic similarity and semantic similarity. We have developed a
query recommendation system that applies two types of searches for repository queries—
semantic search, which finds queries that were written for the same task as the user
query, and syntactic search, which finds queries that are syntactically similar to the user
query.

To illustrate the need for these two types of search and the difference between them,
consider the following scenario. A user needs to write an SQL query that finds the
identifiers of employees in the “HR” department. He starts by formulating a simple
“incorrect” query:

```
SELECT emp_id FROM Employees, Departments
  WHERE dept_name = 'HR'
```

Then, the following repository query, can assist formulating a correct query:

```
SELECT emp_id FROM dept_emp NATURAL JOIN Departments
  WHERE dept_name = 'Accounting'
```

We consider this query as a recommendation, and we want the system to return it, or
other similar queries, to the user. From seeing the recommended repository query, the
user can learn that the identifiers of employees who work in a specific department can be
retrieved using the table `dept_emp`. The user can now change the value “Accounting” to “HR”, in the recommended query, and accordingly, formulate a correct query.

Now, consider the following scenario as an illustration of **semantic search**. A user wants to write a query that finds the identifiers of those people who never purchased an item whose cost exceeds 100 dollars. The user writes the following query:

```
SELECT personid FROM Purchases
  GROUP BY personid HAVING max(cost) <= 100
```

This query does not solve the task correctly, because it does not return the identifiers of the people who never purchased a single item. For the user, it will be beneficial to see the following repository query, which was written for the same task:

```
SELECT personid FROM People WHERE personid NOT IN
  ( SELECT personid FROM Purchases WHERE cost > 100 )
```

In this case, the user may choose to run the repository query instead of the original query, for instance if the recommended query was written by an expert and was tested and optimized.

In order to perform syntactic and semantic search, we need to be able to measure syntactic and semantic similarities between queries. Standard text-search methods such as the vector model [20], TF-IDF [4], or string edit distance [18] are not well suited for measuring semantic and syntactic similarity between SQL queries. Text-search methods are inappropriate for evaluating semantic similarity since minor changes in a query may result in significant change in the semantics of the query. Standard text comparison methods ignore the syntactic structure of compared texts, but as we show later, the syntactic structure is crucial for measuring syntactic similarity between queries.

In this work, we present a method based on Tree Edit Distance to measure the syntactic similarity between queries and a method based on Result Analysis to measure the semantic similarity between SQL queries. These methods are accurate but computationally expensive. Thus, we present two efficient but less effective adaptations of existing methods, namely Attribute Analysis and Vector Model. These two adaptations of known methods are computationally cheaper than the first two methods and have a low number of false negative errors. Hence, we use them to reduce the number of queries that are compared by the more expensive methods. We also use the Attribute Analysis and Vector Model methods as a benchmark to illustrate the accuracy of the other methods.

The Tree Edit Distance method, for measuring syntactic similarity, is a modification of an existing method that compares syntactic structures as ordered trees. The Tree Edit Distance method calculates the edit distance between the syntax trees of the queries to evaluate the similarity between queries. In this work we show that syntax trees of SQL queries cannot be compared as ordered trees. However, computation of the tree edit distance between unordered trees is NP-hard. Thus, we present in this work a heuristic
which is a modification of the ordered-tree edit distance algorithm presented in [27], to deal with unordered trees.

We developed the Result Analysis method for measuring semantic similarity between SQL queries. It is important to note the difference between semantic similarity, as we define it, and query equivalence. Query equivalence is a computationally hard problem, in the general case, even if we restrict the search to conjunctive queries, since for conjunctive queries, query equivalence has the same complexity as the graph-isomorphism problem [8], which is in NP but is not known to be in P, (although it is unknown whether it is an NP-hard problem [13]). The Result Analysis method does not discover equivalence between queries. The method is based on the idea that when two queries are written to perform the same task, they are expected to yield similar results over almost any database. Thus, we generate database samples and during the search we estimate similarity by comparing the results of the queries on the samples. A challenge we had to tackle is how to generate database samples that will not be too large and yet will provide an accurate similarity estimation.

The Vector Model method is an adaptation of the standard IR technique for comparing SQL queries. This method presents queries as vectors of terms and uses similarity between vectors to evaluate the similarity between queries. The traditional vector model method is not suitable for comparing SQL queries since most SQL queries contain the same set of reserved words. Thus, we had to modify the method. We extract terms that are relevant for comparing SQL queries and merge relevant groups of adjacent words into extended terms.

The Attribute Analysis method is a variation of a method presented in [3]. The principle idea is that two queries are similar if they deal with similar sets of attributes. In order to compare two queries, we compare the sets of attributes that are used in the constraints of the queries and the sets of attributes that are only projected in the queries. The Attribute Analysis method is computationally inexpensive, since the sets of relevant attributes are usually small. The drawback of this method is a high number of false positive errors, since it considers any queries that operate on the same sets of attribute, as similar, while ignoring the role of the attributes or where they appear in the queries.

In addition to the search for repository queries, where a user provides his own query and receives recommended queries that are similar syntactically or semantically to his query, we have developed two additional types of searches that do not require the user to provide an initial query. In the syntactic guided search, the user searches for repository queries using fragments of a query as the search keys. In the semantic guided search, the user searches for repository queries by specifying the expected result of the query.

We have collected a set of real queries and performed experimental evaluation to test the accuracy of all the methods, for both the syntactic and the semantic similarity measures.

The thesis is organized as follows. Chapter 2 provides an overview of related work.
Chapter 3 presents formal definitions of the search problems, definitions of the two types of similarities between queries and additional definitions and notations that will be used in the thesis. Chapter 4 depicts the four methods that we developed for measuring syntactic and semantic similarity. Chapter 5 shows how to conduct syntactic guided search and the semantic guided search. Chapter 6 presents our experimental evaluation and it shows that the Tree Edit Distance has the best accuracy for measuring syntactic similarity and that the Result Analysis method has the best accuracy for measuring semantic similarity. Finally, in Chapter 7 we conclude.
Chapter 2

Related Work

The problem of query recommendation received some attention recently. The importance of a query recommendation tool as part of a collaborative large-scale data management system has been emphasized by Khoussainova et al. [16]. They claim that database management systems could greatly benefit from query recommendation capabilities. When users continuously write new queries, these users should have the ability to leverage knowledge about queries that they or others have previously executed on the data. Khoussainova et al. [26] presented a framework for collaborative query management systems to be used by collaborative databases, to allow users apply exploratory queries and receive query recommendations. It has been asserted that even though the problem of generating query recommendations has been broadly addressed in the context of the World-Wide Web, only a handful of related papers deal with query recommendation in the database context [2].

Danaparamita and Gatterbauer [10] presented a system that helps users understand the meaning of SQL queries by creating a graph that illustrates the intent of the given SQL query. Using their system, users can browse through sets of repository queries, understand their patterns, and leverage those query templates to compose new queries. However, this system is not practical for the type of search we described in the introduction. Other systems were developed to assist the SQL composition process by suggesting auto-completion [17] or automatic query generation [6], but these systems do not apply search to find a query in a repository of queries.

2.1 Session-Based Recommendations

A query recommendation problem similar to the problem that we address is how to recommend queries based on the analysis of a user session [2, 7, 24]. Session-based recommendation is based on a simple premise that is inspired by Web recommendation systems: If user $U_1$ and user $U_2$ have similar querying behaviors, then they are likely interested in retrieving the same data. Hence, the queries of user $U_2$ can serve as a guide for user $U_1$. So, the problem is to find a recorded session of queries, that is the most
similar to the current user session, in order to recommend queries from the discovered session. This problem is different from the problem we are studying, since we do not rely on sessions or on the existence of sequences of queries, to conduct the search.

There are two approaches for measuring the similarity between sessions: a tuple-based approach and an item-based approach. Chatzopoulou et al. [7] developed a query recommendation system that employs a tuple-based approach to measure the similarity between sessions. They concentrate on Select-Project-Join queries, because such queries are common in interactive database exploration, particularly among novice users. They consider two sessions as similar if the queries of the sessions “touch” the same tuples, i.e. depend on similar sets of tuples. A query touches a tuple if the tuple contributes at least one row to the result of the query. However, both [24] and [7], limit their study to Select-Project-Join queries. More importantly, they require having the results of the user queries for applying their method, and this is unsuitable for a general search where the user query can be partial or incorrect.

Akbarnejad et al. [2] present an item-based approach for session-based recommendation. Two sessions are considered similar if their queries have similar syntactic fragments. The syntactic fragments are, for example, WHERE clauses that contain only attribute names, relaxed constant values and relaxed predicates. In the relaxation, some tokens are replaces by constants, e.g., all strings, hexadecimal numbers and decimal numbers are replaced by STR, HEXNUM and NUM, respectively. A similar generalization is also applied on lists or ranges of numbers and strings. The mathematical and set comparators are also replaced by string equivalents, e.g., “=” is replaced by EQU and “≤” by COMPARE. Then, the similarity between two queries is measured by the number of common relaxed syntactic fragments. This approach that is justified in the context of session comparison is not accurate enough for comparison of the queries themselves. Two queries may have the same relaxed syntactic fragments and still be different by any means.

2.2 Query Clustering

The problem of comparing SQL queries is also relevant in other recommendation scenarios. In [14], query clustering is conducted based on a comparison of query structures, the associated table schemes and statistics such as the sizes of the tables that appear in the queries. Their motivation of clustering is to group queries based on the similarity of their execution plan, which is very different from semantic or syntactic similarity. Hence, this approach is not suitable for the search problem that we investigate.

Using sets of features for query clustering has also been presented for context-based recommendation [9]. The motivation here is to add relevant information to the queries even when it is not explicitly asked, when it is of interest to the user. This approach compares queries using a predefined semantic model, and differently from us, relies on the context of the queries. Aouiche et al. [3] presented an approach where queries
are clustered merely based on their attributes, for materialized view selection. They suggested measuring similarity between queries and also similarity between sets of queries based only on the attributes that were extracted from the queries, but this approach is not accurate enough for general purposes. None of these papers provides a query similarity metric in the sense that we suggest. Hence, previously presented approaches are inappropriate for the search tasks we consider. Furthermore, context-based approaches use a simplified query similarity that is reduced to similarity between relaxed fragments. This alone cannot provide out-of-context query similarity metric, because the similarity evaluation produced by relaxed fragments is too rough. A different approach of feature-based clustering has been presented in [14]. Two queries are similar if they have the same work plan, although they may not have the same semantic. Obviously, this is different from our approach.

In [1, 12, 30] the authors present systems that analyze common subexpressions of queries in order to reuse the results of historical queries for optimization. This problem is different from our problem since our goal is to compare entire queries and not only subexpressions of queries.

The problem of query recommendation is an important problem, as was emphasized in [16]. The majority of the papers on query recommendation present session-based approaches for recommending queries. Consequently they utilize basic techniques for comparing queries, since it is easier to compare two sets of queries than two queries out-of-context. The problem of comparing SQL queries has been addressed not only in the context of query recommendation. Queries are also compared to predict the execution plan or for materialized view selection, for optimization purposes [1, 12, 30]. The approaches that were applied in those papers are not suitable for the type of search we consider since they were not developed to handle syntactic or semantic similarity between queries.
Chapter 3

Framework

This chapter presents our general model and formal definitions of the two types of SQL query search. We will also present some notations that will be used in the following sections of the thesis.

3.1 Query Search

In this study, we consider the problem of query search, where for a given user query $Q_u$ and a repository of queries $\{Q_1, \ldots, Q_n\}$, the goal is to return the repository queries that are similar, in a sense that will be defined later, to the user query. The queries we refer to in this study are standard SQL queries, according to the syntax supported by the SQLite database management system [23]. We distinguish between two types of similarities between SQL queries: syntactic similarity and semantic similarity.

3.1.1 Syntactic Search

In syntactic search, the goal is to find repository queries that are syntactically similar to the user query. In this section, we define the notion of syntactic similarity, then we formally define syntactic search and we discuss cases in which it is beneficial to be able to measure syntactic similarity between queries.

To illustrate our approach, consider the following scenario. Suppose a user has written an SQL query and wants to know if queries that look the same as her query have already been written by other users. We assume that previously written queries are stored in the log and we refer to them as repository queries or log queries. The user may want to see these similar queries to gain knowledge about the database—knowledge that will help her formulate more complicated queries. Such similarity may also allow her to trace cases of duplication, e.g., in the homework assignments of a course, two similar queries may indicate the lack of academic integrity, i.e., it will alert that one homework is perhaps copied from the other. In this case, the goal is to find repository queries that are similar to the user query by the way they are written and not necessarily queries that were written to perform the same task as the user query.
**Definition 3.1.1 (Syntactic Similarity).** Two SQL queries are syntactically similar if they are considered similar by users who only examine the way they are written, without taking into account the underlying task for which they could have been written.

Syntactic search is based on syntactic similarity.

**Definition 3.1.2 (Syntactic Search).** Syntactic search is the following problem: Given an SQL query $Q_u$ and a repository of queries $\{Q_1, \ldots, Q_n\}$, find the repository queries that are syntactically similar to $Q_u$.

Note that syntactic similarity can also be used in tasks such as clustering queries, where queries that are syntactically similar are grouped. This may help presenting to users large sets of queries in a succinct or generalized manner. Presenting merely a single query for each group of syntactically similar queries can be useful when presenting the results of any query search, especially of searches different from syntactic search, since it will prevent duplications in the result. Also note that the ability to find syntactically similar queries improves the ability to measure the similarity between user sessions—the problem that was investigated in [24] and [7].

### 3.1.2 Semantic Search

We now formally define the notion of *semantic similarity* and use this notion to define *semantic search*. When two users write SQL queries to perform the same task they do not necessarily write identical or even syntactically similar queries, even when the queries are posed over the same database. In many scenarios it is beneficial to be able to find pairs of queries that were written for the same task. For example, to leverage a query written by an expert for the same task. This may also assist database administrators detect cases where queries of specific types are posed, e.g., when these queries may indicate a misuse of the data. Thus, we define semantic similarity and semantic search as follows.

**Definition 3.1.3 (Semantic Similarity).** Two SQL queries are semantically similar if they could have been written to perform the same task.

**Definition 3.1.4 (Semantic Search).** Semantic search is the following problem: Given an SQL query $Q_u$ and a repository of queries $\{Q_1, \ldots, Q_n\}$, find the repository queries that are semantically similar to $Q_u$.

As with syntactic similarity, also semantic similarity can be used for clustering queries or for monitoring the system, to detect cases where a query that performs a certain task has been posed to the database.
3.1.3 Semantic Similarity versus Syntactic Similarity

It is important to note that semantic similarity does not entail syntactic similarity, and vice versa. There are semantically dissimilar queries that are syntactically similar and there are syntactically dissimilar queries that are semantically similar. For example the following two queries are syntactically similar.

```
SELECT * FROM People NATURAL JOIN Graduate
ON Graduate.year = 1988

SELECT * FROM People NATURAL JOIN Graduate
ON Graduate.year != 1988
```

These queries look very similar, especially in the eyes of a layman, since they are posed over the same tables, using the same attributes and with constraints that syntactically are quite similar. Yet, these two queries are semantically dissimilar, since obviously they were written to perform different tasks.

To illustrate the other direction, the following two queries are semantically similar.

```
SELECT * FROM People NATURAL JOIN Graduate
ON Graduate.university != 'Technion'

SELECT * FROM People JOIN
(SELECT * FROM Graduate EXCEPT
(SELECT * FROM Graduate WHERE university = 'Technion'))
ON People.id = Graduate.id)
```

One can see that they were written to perform the same task, that is, to find the list of people graduated a university other than the Technion. However, these two queries are not syntactically similar, since they have different structures and use different keywords. A layman would not be able to detect the similarity between these two queries.

3.2 Additional Definitions and Notations

In this section we present some basic notions and notations that will be used in the following sections.

Syntactic similarity can be tested using the syntax trees of SQL queries. Since syntax trees are constructed in the usual way, we illustrate them and avoid providing the exact construction procedure. To illustrate syntax trees, consider the following query.

```
SELECT x FROM A WHERE x > 0 AND x < 10
```

This query has the following syntax tree.
Sub queries are represented using subtrees. So, the WHERE part of the following query

\[
\text{SELECT } x \text{ FROM } A \text{ WHERE } x \text{ in (SELECT } x \text{ FROM } B)\\
\]

is represented by the following syntax tree:

\[
\text{WHERE} \\
| \text{in} \\
| x \text{ subquery} \\
SELECT \text{ FROM} \\
| x \text{ } | B
\]

The syntax trees of SQL queries are rooted ordered labeled trees. Rooted ordered labeled trees are defined in the usual way, e.g., see [11]. Note that the following definition is recursive.

**Definition 3.2.1** (Rooted Ordered Tree). A rooted ordered tree \( T \) is either a single node \( r \) or a node \( r \) whose children are an ordered sequence of roots of disjoint rooted ordered trees. The node \( r \) is the root of \( T \). We denote by \( T = r(A_1 \circ \cdots \circ A_n) \) a tree \( T \) that has a root \( r \) whose children are the roots \( a_1, \ldots, a_n \) of the trees \( A_1, \ldots, A_n \).

Given a rooted ordered tree, we denote by \( V(T) \) the set of all the nodes of \( T \). We denote by \( |T| \) the number of nodes of \( T \). We denote by \( \text{deg}(v) \) the degree of a node \( v \) in \( T \), i.e., the number of children of \( v \).

$Labeled rooted ordered trees$ are rooted ordered trees with labels assigned to nodes. In SQL syntax trees, the labels on the nodes are names of tables, attributes, reserved words of SQL and constants that appear in the queries.

**Definition 3.2.2.** Given a non-empty set of labels \( \Sigma \), a labeled rooted ordered tree over \( \Sigma \) is a pair \( (T, L) \), where \( T \) is rooted ordered tree and \( L : V(T) \to \Sigma \) is a function that maps each node of \( T \) to a label from \( \Sigma \).

Comparison of labeled rooted ordered trees is based on testing isomorphism between pairs of trees. Intuitively, two labeled rooted ordered trees are isomorphic if they have the same topology and the same labels.
**Definition 3.2.3.** Consider two labeled rooted ordered trees \((T_1, L_1)\) and \((T_2, L_2)\), where \(T_1 = r_1(A_{1,1} \circ \cdots \circ A_{1,\deg(r_1)})\) and \(T_2 = r_2(A_{2,1} \circ \cdots \circ A_{2,\deg(r_2)})\). We say that \((T_1, L_1)\) and \((T_2, L_2)\) are isomorphic if one of the following holds.

1. Both trees have a single node—the root—so that \(\deg(r_1) = \deg(r_2) = 0\) and \(L_1(r_1) = L_2(r_2)\).

2. The following three conditions hold.
   
   (a) The roots of the trees have the same label: \(L_1(r_1) = L_2(r_2)\).
   
   (b) The roots have the same number of children: \(\deg(r_1) = \deg(r_2)\).
   
   (c) The children of the roots are pairwise isomorphic: \((A_{1,i}, L_1)\) is isomorphic to \((A_{2,i}, L_2)\), for all \(1 \leq i \leq \deg(r_1)\).

When considering the semantics of queries, it is needed to compare their results. The results of queries are *multisets*, i.e., duplicate tuples are not eliminated unless elimination is explicitly requested, since SQL has bag-theoretic semantics. A multiset (also known by the term *bag*) is a generalization of a set by allowing elements to appear more than once. In this work we formally define and denote multisets as follows.

**Definition 3.2.4 (Multiset).** The multiset is a 2-tuple \((S, m)\) where \(S\) is the underlying set and \(m : S \to \mathbb{N}_{>0}\) is a multiplicity function assigning each element to its number of appearances in the multiset.

The intersection of two multisets \((S_1, m_1)\) and \((S_2, m_2)\) is a multiset \((S, m)\) such that \(S = S_1 \cap S_2\) and for each \(a \in S\), holds \(m(a) = \min(m_1(a), m_2(a))\). The cardinality of a multiset \((S, m)\) is denoted \(|(S, m)|\) and it is equal to \(\sum_{a \in S} m(a)\), i.e., the number of elements in \(S\), with repetitions.

In order to compare sets and multisets we use Jaccard set and multiset similarity metric. For sets, it is defined as follows.

**Definition 3.2.5 (Jaccard Similarity for Sets).** Let \(S_1\) and \(S_2\) be two sets. The Jaccard similarity between \(S_1\) and \(S_2\) is

\[
\frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}
\]

For multisets, Jaccard similarity is defined as follows.

**Definition 3.2.6 (Jaccard Similarity for Multisets).** Let \(M_1\) and \(M_2\) be two multisets. The Jaccard similarity between \(M_1\) and \(M_2\) is

\[
\frac{|M_1 \cap M_2|}{|M_1| + |M_2| - |M_1 \cap M_2|}
\]
Note that the expression $|M_1| + |M_2|$ in the multiset version replaces the size of the union in the set version. The expression $\frac{|M_1 \cap M_2|}{|M_1| + |M_2|}$ never equals to 1 for any non-empty multisets $M_1$ and $M_2$. That is why the expression $|M_1 \cap M_2|$ is added, since the metric should return values in the entire $[0, 1]$ range.

We use the standard notions of recall, precision and F-measure to evaluate the quality of the results of the method we developed for measuring similarities between queries.

**Definition 3.2.7.** Given a user query $q_u$, a subset $S_E$ of repository queries that are known to be similar to $q_u$ and a subset $S_A$ of the queries returned as the answer of the search, the recall is

$$\frac{|S_E \cap S_A|}{|S_A|}$$

the precision is

$$\frac{|S_E \cap S_A|}{|S_E|}$$

and the F-measure is the harmonic mean of the recall and the precision, i.e.,

$$\frac{2 \cdot \text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}}$$
Chapter 4

Methods Description

In this chapter we present the methods for evaluation of syntactic and semantic similarities between SQL queries: *Tree Edit Distance, Attribute Analysis, Vector Model* and a *Result Analysis* approaches.

4.1 Tree Edit Distance

In this section we define the Tree Edit Distance method for measuring similarity between queries. Intuitively, this method utilizes the similarity between the syntax trees of the compared queries, to measure the similarity between the queries. Syntax trees are compared by considering them as ordered labeled trees and computing a *tree edit distance* between them. In the following section, we define tree edit distance between labeled ordered trees. To make this thesis self-contained, we include in it a description of an existing algorithm for calculating the edit distance between ordered labeled trees (OTED)—the algorithm of Zhang and Shasha [27]. Then, we explain why syntax trees of SQL queries cannot be considered as ordered labeled trees, and why the algorithm for order trees does not work for unordered trees. Since computing the tree edit distance for unordered labeled trees is NP-hard, we present an effective heuristic for this problem.

4.1.1 Edit Operations and Tree Edit Distance

In order to define the tree edit distance we first define edit operations on ordered labeled trees, as they were defined in [25]. There are three types of operations: label substitution, node deletion and node insertion.

**Definition 4.1.1** (Label Substitution). Label substitution by changing $l_1$ to $l_2$ on node $v$, in an ordered labeled tree $(T, L)$, results in a tree $(T, L')$, where $L'(u) = L(u)$ for all $u \neq v$ and $L'(v) = l_2$.

Figure 4.1 shows the substitution of label $l_1$ by $l_2$ for node $v_3$. Note that other nodes with label $l_1$ remain unmodified.
Deletion of a node $v$ means making the children of $v$ become the consecutive children of the parent of $v$ and then removing $v$.

**Definition 4.1.2** (Node Deletion). Let $(T, L)$ be a rooted ordered labeled tree with node $p$, and let the sequence $a_1, \ldots, a_{\text{deg}(p)}$ be the children of $p$. Deletion of child $a_i$ that is the parent of the sequence of nodes $c_1, \ldots, c_{\text{deg}(a_i)}$ changes the sequence of children of $p$ to be $a_1, \ldots, a_{i-1}, c_1, \ldots, c_{\text{deg}(a_i)}, a_{i+1}, \ldots, a_{\text{deg}(p)}$. The rest of the tree and the labeling of the rest of the nodes remain unchanged.

Figure 4.2 shows deletion of a node $v$ which is a child of node $v_1$. Since the labeling of the nodes is not affected by the deletion, we do not present the labels in the figure.

Insertion is the complement of deletion. This means that inserting $v$ as a child of $p$ will make $v$ the parent of a consecutive subsequence of the current children of $p$.

**Definition 4.1.3** (Node Insertion). Let $(T, L)$ be a rooted ordered labeled tree with node $p$ such that the sequence $a_1, \ldots, a_{\text{deg}(p)}$ are the children of $p$. Insertion of a node $v$ labeled $l$ as the parent of the $k$ consecutive children of $p$, starting from $a_i$, (that is, $a_i, \ldots, a_{i+k-1}$) changes the sequence of children of $p$ to be $a_1, \ldots, a_{i-1}, v, a_{i+k}, \ldots, a_{\text{deg}(p)}$, where the children of the new node $v$ are $a_i, \ldots, a_{i+k-1}$. The rest of the tree and the labeling of the rest of the nodes remain unchanged.

Figure 4.3 shows the insertion of a node $v$ as a parent of $v_3, v_4$. Since the labeling of the nodes is not affected by the insertion, we do not present the labels in this figure.
The three edit operation—node deletion, node insertion and label substitution—provide a complete set of edit operations, in the sense that these three operations are sufficient to transform any ordered labeled tree into any other ordered labeled tree. This can be done, for example, by deleting all the nodes of the original tree, except for the root, changing the label of the root to be the label of the root of the target tree and inserting all the nodes (except for the root) of the target tree.

Ordered labeled tree edit distance is defined using the tree edit operations. The definition is similar to the definition of tree edit distance in [25].

**Definition 4.1.4.** Let $t_1$ and $t_2$ be ordered labeled trees and let $S$ be the set of all sequences of edit operations (substitution, insertion and deletion) that transform $t_1$ into a tree that is isomorphic to $t_2$. Let $|s|$ denote the number of edit operations in sequence $s$. Then, the ordered tree edit distance, denoted $OTED$ is defined as $OTED(t_1,t_2) = \min_{s \in S} |s|$.

In order to calculate the ordered tree edit distance between syntax trees we use the $OTED$ algorithm of Zhang and Shasha [27]. The essence of the algorithm is explained in Section 4.1.3. This algorithm uses dynamic programming to compute the ordered tree edit distances between $t_1$ and $t_2$.

Similarity is opposite of distance—high distance between the trees means low similarity, and vice versa. Also, we want the similarity measure to be a real number between 0 and 1, whereas the tree edit distance is a number between 0 and $|t_1| + |t_2|$. Thus, we define the similarity measure that is based on tree edit distance, namely $TED$ similarity, by using $OTED$ as follows:

$$sim_{TED}(t_1,t_2) = 1 - \frac{OTED(t_1,t_2)}{|t_1| + |t_2|}.$$ 

So, when $t_1$ and $t_2$ are identical, $OTED(t_1,t_2)$ is 0, and then the similarity is $1 - \frac{0}{|t_1| + |t_2|} = 1$. When $t_1$ and $t_2$ are different trees, then $OTED(t_1,t_2)$ is at most $|t_1| + |t_2|$—the cost of deleting all the nodes of $t_1$ and inserting all the nodes of $t_2$—and then the similarity is at least 0:

$$0 = 1 - \frac{|t_1| + |t_2|}{|t_1| + |t_2|} \leq sim_{TED}(t_1,t_2) \leq 1 - \frac{0}{|t_1| + |t_2|} = 1.$$
4.1.2 Syntax Trees are not Ordered Trees

In this section we explain why we cannot consider syntax trees as ordered trees, when measuring the similarity between queries, and we present our solution for this problem.

Syntax trees of SQL queries are not ordered trees since there are nodes in the syntax trees for which the order of the children is unimportant, when considering the semantics of the query. To illustrate this, consider the two subtrees presented in Figure 4.4.

![Figure 4.4: Similar subtrees that are dissimilar as ordered trees.](image)

The two subtrees in Figure 4.4 are similar one to the other, however, the OTED between these subtrees is equals to twice the OTED between $T_1$ and $T_2$, and if $T_1$ is very different from $T_2$, this distance can be big. Since the order of the children of the node labeled AND is unimportant, the real distance should be calculated between the subtrees in Figure 4.5.

![Figure 4.5: Similar subtrees that are also similar as ordered trees.](image)

The distance between the two subtrees in Figure 4.5 is equal to the cost of label substitution from $a$ to $b$ plus label substitution from $b$ to $a$, which in most cases is much smaller than $OTE_D(T_1, T_2)$, since $T_1$ and $T_2$ can be very different trees. This example shows that if we calculate the distance between syntax trees as if it is the distance between ordered labeled trees, we will have useless and incoherent results, e.g. where queries that are practically the same are far from one another.

While comparing syntax trees using unordered tree edit distance seems a good idea, unordered tree edit distance is an NP-hard computation problem [27]. In fact, the problem is MAX SNP-hard, which means that there is no polynomial time constant ratio approximation to this problem [28]. Adding a reordering edit operation also does not simplify the problem. For example, one can add, to the set of edit operations, an operation that swaps two adjacent children of some node. But the problem of computing the tree edit distance between ordered trees with the swap operation included is also an NP-hard problem [5].
Since the problem of computing the unordered tree edit distance between two trees \( t_1 \) and \( t_2 \) in NP-hard, we use a heuristic to examine different orders of the children of order-indifferent nodes of \( t_1 \) and \( t_2 \). The heuristic strives to minimize the ordered tree edit distance between \( t_1 \) and \( t_2 \). A node in a syntax tree is considered an order-indifferent node if its children can be reordered without changing the semantics of the query. Examples of such nodes are nodes labeled \textit{AND}, \textit{JOIN} and \textit{UNION}. Typically, the number of ordered-indifferent nodes in a syntax tree of a real query is small, and in particular, it is much smaller than the total number of nodes in the syntax tree.

We use a greedy heuristic search to find the order for the children of the order-indifferent nodes, in both \( t_1 \) and \( t_2 \), where the goal is to find the order that minimizes \( OTED(t_1, t_2) \). In each iteration, the search selects one order-indifferent node of \( t_1 \) or \( t_2 \) and sorts its children to achieve the maximal decrease in the \( OTED \) between \( t_1 \) and \( t_2 \). The search stops when it does not find any node that can be reordered to decrease the \( OTED \) between \( t_1 \) and \( t_2 \). Each iteration of the search reduces the tree edit distance between the trees, by at least 1. That is why the search always stops by finding a local minimum of the ordered tree edit distance between the trees.

The search finds a local maximum of the TED similarity (the local minimum of the distance). Note, however, that the local maximum can be different from the global maximum. To illustrate how often the search does not find the global maximum, we briefly present some experimental results (the description here is partial—detailed experiments are presented in Chapter 6). We performed the following experiment for a set of real queries with order-indifferent nodes. Given a query \( q \) we parsed the query and extracted the query syntax tree \( t \). We randomly changed the order of all the order-indifferent nodes of \( t \), to create another syntax tree \( t' \). Clearly, the tree edit distance between \( t \) and \( t' \) is 0 and the similarity between them is 1, since we only permute the children of order-indifferent nodes. We compared \( t \) and \( t' \) as ordered trees using the original tree edit distance algorithm of Zhang and Shasha [27]. The average similarity between such trees was 0.8. We compared \( t \) and \( t' \) using our greedy heuristic search and the average similarity between the trees was 0.97.

This experiment shows that the greedy search is better in discovering similarity between syntax trees than the ordered tree edit distance algorithm. Generally, the local maximum search can be performed multiple times starting from different random orders of the children of the order-indifferent nodes, to increase the chances of finding the global maximum. We can see from the experiment that on the average, another invocation of the search will not improve the quality of the computed measure by more than 3%, but it will double the running time of the method. That is why in the search for the global maximum we conduct only a single invocation of the search.

Our study of cases where the global maximum is not discovered revealed that the most common scenario of a local maximum is of cases similar to the case depicted in Figure 4.6.
The tree edit distance between the trees in Figure 4.6 is 0, whereas the ordered tree edit distance between these trees is 4. In this example, the ordered tree edit distance cannot be decreased by merely changing the order of the children of a single node. Thus, the heuristic search cannot decrease the ordered tree edit distance to be lower than 4, although the real distance is 0. In practice, subtrees similar to the subtrees in Figure 4.6 appear close to the leafs of the entire syntax trees, and thus, the influence of such local maxima is not significant. This explains why the accuracy of the search is very high, according to our experiments.

4.1.3 Original TED and Heuristics Search Algorithms

We now present the original OrderedTED algorithm of [27] followed by the the heuris-
tic search over orders of order-indifferent nodes algorithm and finally we present the improvement of the original OrderedTED algorithm that allows reusing dynamic data between multiple computations of OrderedTED algorithm.

Definitions

We start by providing some definitions and notations that are needed for the presentation of the ordered tree edit distance algorithm.

We number the nodes of a given tree, based on a left-to-right postorder numbering. Let $T[i]$ be the $i$-th node of a tree $T$ according to the left-to-right postorder numbering. Let $T[i..j]$ be the ordered subforest of $T$ induced by the nodes numbered from $i$ to $j$ inclusive according to the left-to-right postorder numbering. When it is clear from the context, we omit the $T$ and refer to the node as $i$. The index of the parent of $T[i]$ is denoted $parent(i)$. We define $parent^0(i) = i$, $parent^1(i) = parent(i), parent^2(i) = parent(parent(i))$, and so on. We consider $depth(i)$ to be the length of the path from the root of $T$ to $i$. The set of ancestors of a node $i$ in $T$ are all the nodes on a path from the root to $i$, including $i$ itself. We denote this set by $ancestors(i)$, and formally define it as $ancestors(i) = \{parent^k(i) \mid 0 \leq k \leq depth(i)\}$.

Let $lm$-leaf$(i)$ be the leftmost leaf descendant of the subtree rooted at $T[i]$. When $T[i]$ is a leaf, $lm$-leaf$(i) = i$. The set KeyRoots$(T)$ comprises the root of $T$ and all the nodes that have a left sibling. For a node with a left sibling, its parent has a leftmost leaf descendant different from its leftmost leaf descendant. For a node without a left sibling, it has the same leftmost leaf descendant as all of its ancestors. Thus, we
define \( \text{KeyRoots}(T) = \{ i \mid (i = 0) \vee (\exists i' \in \text{ancestors}(i) : \text{lm-leaf}(i) \neq \text{lm-leaf}(i')) \} \).

Intuitively, this set consists of the roots of all the subtrees of \( T \) for which there is a need to compute the distances separately from the other subtrees. The distance between the subtree rooted at \( T[i] \) and the subtree rooted at \( T[j] \) is denoted \( \text{sted}(i, j) \).

**Basic Ordered Tree Edit Distance Algorithm**

We present now a basic algorithm that uses dynamic programing to compute the distance between subtrees of \( t_1 \) and \( t_2 \). The algorithm consists of the main loop, presented in Figure 4.7, and the inner loop, presented in Figure 4.8. The final iteration of the main loop computes the ordered tree edit distance between \( t_1 \) and \( t_2 \).

```
\text{OrderedTED}(t_1, t_2) \\
K_1 \leftarrow \text{KeyRoots}(t_1) \\
K_2 \leftarrow \text{KeyRoots}(t_2) \\
\text{Sort } K_1 \text{ and } K_2 \text{ in increasing order} \\
\text{for } i = 1 \text{ to } |K_1| \text{ do} \\
\hspace{1em} \text{for } j = 1 \text{ to } |K_2| \text{ do} \\
\hspace{2em} t_i \leftarrow K_1[i] \\
\hspace{2em} t_j \leftarrow K_2[j] \\
\hspace{2em} \text{Compute } \text{sted}(t_i, t_j) \\
\hspace{1em}\text{end for} \\
\text{end for} \\
\text{return } \text{sted}(|t_1|, |t_2|)
```

**Figure 4.7:** A main loop of the basic algorithm, similar to the algorithm of [27], to compute \( \text{OTED}(t_1, t_2) \).

The \( \text{forestdist} \) values computed and used in Algorithm 4.8 are put in a temporary array that is freed once the corresponding treedist is computed. The \( \text{sted} \) values are put in the permanent array. Since the cost of all the edit operations is uniform, the cost of the insertion, deletion and substituition was substituted by 1 in the inner loop part of the original algorithm in Figure 4.8.
The pseudo code of the search is presented in Figure 4.9. The function OrderedTED finds the tree edit distance for ordered trees and it is an invocation of the algorithm of Zhang and Shasha [27]. We will discuss this algorithm in Section 4.1.3. The function SelectNodeToSort finds the node whose children should be reordered in the current iteration. The function returns the first node for which reordering of its children decreases OTED(t₁, t₂). The order by which the function checks the unordered nodes may affect the result of the algorithm. Intuitively, reordering bigger subtrees changes the OTED more significantly. Thus, in out heuristic, the order by which the function checks the unordered nodes is from the node with the largest number of decedents to the node with the smallest number of decedents (i.e., starting with the order-indifferent node that is the root of the largest subtree). The function SelectNodeToSort returns a node that has at least a pair of adjacent children whose swap decreases the OTED or returns null if no such node exists. Its code is presented in Figure 4.13. This method iterates over the nodes U of the two trees and checks for each node n if a swap of two adjacent children i − 1 and i of n, decrease the distance between the trees.

For example consider the two ordered labeled trees T₁ and T₂ rooted at u₁ and v₁, re-
SearchTED($t_1, t_2$)

Preprocessing:
let $U \leftarrow \text{Nodes-of}(t_1) \cup \text{Nodes-of}(t_2)$

sort the nodes in $U$ by their number of descendents, decreasingly

$d \leftarrow \text{OrderedTED}(t_1, t_2)$

$prev \leftarrow U[0]$

loop

$n \leftarrow \text{SelectNodeToSort}(t_1, t_2, U, prev, d)$

if ($n = \text{null}$) then

return $\text{OrderedTED}(t_1, t_2)$

else

prev $\leftarrow n$

if $\text{deg}(n) > 2$ then

$	ext{ChildrenReordering}(n, t_1, t_2, d)$

end if

end if

end loop

Figure 4.9: Heuristic search over the orders of order-indifferent nodes of $t_1$ and $t_2$, to find the order with the minimal $\text{OTED}(t_1, t_2)$.

spectively, as depicted in Figure 4.10. Let $\text{OTED}(u, v)$ be the tree edit distance between the subtrees of $T_1$ and $T_2$ rooted at $u$ and $v$. Then, $\text{OTED}(T_1, T_2) = \text{OTED}(u_1, v_1) = 6$, since $\text{OTED}(u_2, v_2) = 3$ and $\text{OTED}(u_5, v_6) = 3$. Now assume that all the nodes whose label is either root or a are order-indifferent nodes. The order-indifferent nodes of $T_1$ and $T_2$ sorted by the number of descendents, decreasingly, are $u_1, v_1, u_5, v_2, u_2, v_6$. The function $\text{SelectNodeToSort}$ selects nodes in this order. In each iteration, the function $\text{SelectNodeToSort}$ starts with the node that precedes the node whose children were permuted in the previous iteration, where after the last node it proceeds to the first node, i.e., after processing $v_6$ it continues with $u_1$. This means that if in some iteration the children of node $u_5$ were reordered, then in the following iterations $\text{SelectNodeToSort}$ will return $v_2, u_2, v_6, u_1, v_1, u_5, \ldots$

Table 4.1 shows the order by which the nodes were checked and reordered during the search. In the first iteration of $\text{SearchTED}$, the function $\text{SelectNodeToSort}$ selects $u_1$ as the node whose children should be permuted, since $\text{OTED}(T_1, T_2) = 5$ after swapping $u_2$ and $u_5$. Note that $u_1$ was the first node that $\text{SelectNodeToSort}$ checked, since it has
Table 4.1: Execution of SearchTED on $T_1$ and $T_2$—the trees depicted in Figure 4.10.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>$OTED(T_1, T_2)$</th>
<th>Checked nodes</th>
<th>Reordered node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>$u_1$</td>
<td>$u_1$</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>$v_1, u_5$</td>
<td>$u_5$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$v_2, u_2$</td>
<td>$u_2$</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>$v_6, u_1, v_1, u_5, v_2$</td>
<td>None</td>
</tr>
</tbody>
</table>

the largest number of descendents.

In the second iteration of SearchTED, the function SelectNodeToSort checks $v_1$ and $u_5$ and selects $u_5$ as the node whose children should be permuted. Note that the function SelectNodeToSort skips checking $u_1$ since it was reordered in the previous iteration and reordering the children of $v_1$ does not decrease $OTED(T_1, T_2)$. The function SelectNodeToSort swaps $u_6$ and $u_7$, and since $u_5$ has more than 2 children the function ChildrenReordering is used for further reordering of the children of $u_5$.

In this example, the function ChildrenReordering does nothing since the only possible additional swap between $u_6$ and $u_8$ does not decrease $OTED(T_1, T_2)$. At the end of the second iteration, the trees are ordered as in Figure 4.11.

![Figure 4.11: The ordering of $T_1$ and $T_2$ after the second iteration. $OTED(T_1, T_2) = 3.$](image)

In the third iteration of SearchTED, the function SelectNodeToSort checks $v_2$ and $u_2$ and selects $u_2$ as the node whose children should be permuted. After swapping $u_3$ and $u_4$, $OTED(T_1, T_2) = 1$. The last iteration checks the nodes $v_6, u_1, v_1, u_5, v_2$ and does not reorder any node, since no permutation decreases $OTED(T_1, T_2)$. This final order is depicted in Figure 4.12. Figure 4.13 presents the pseudo code of SelectNodeToSort.

![Figure 4.12: The order of the nodes of $T_1$ and $T_2$ after the last iteration. $OTED(T_1, T_2) = 1$, because the single edit operation of changing $c$ to $f$ on $u_8$ will produce isomorphic trees.](image)
**SelectNodeToSort**($t_1, t_2, U, \text{prevNode}, \text{prevDistance}$)

$p \leftarrow U.indexOf(\text{prevNode})$

$N \leftarrow |U|$

for $j = 1$ to $N$

$n \leftarrow U[(p + j) \mod N]$

for $i = 1$ to $\deg(n)$

swap($n, i - 1, i$)

if $\text{UpdateOrderedTED}(t_1, t_2) < \text{prevDistance}$ then

return $n$

end if

swap($n, i, i - 1$)

end for

end for

return null

**Figure 4.13:** Selecting the node for children permutation. The $\text{prevOTED}$ parameter is the value of $\text{OTED}(t_1, t_2)$ from the previous iteration of $\text{SearchTED}$ (see Figure 4.9).

The function $\text{ChildrenReordering}$ finds for a given node $n$, the order of children that minimizes the $\text{OTED}$ between $t_1$ and $t_2$. To increase the efficiency, the function does not check all the possible orders. It performs an iterative search where in each iteration it swaps a pair of adjacent children that decrease the $\text{OTED}$ between the trees. If no such pair exists, it ends the search. This method always terminate because every iteration decreases the $\text{OTED}$ by at least 1. Figure 4.14 presents the pseudo code of the function $\text{ChildrenReordering}$. It uses the function $\text{UpdateOrderedTED}$, which reuses the dynamic data of the $\text{OrderedTED}$ function to calculate the $\text{OTED}(t_1, t_2)$ after a swap. We discuss this function in Section 4.1.3.
ChildrenReordering\((n, t_1, t_2, \text{prevDistance})\)

\[
\text{minDistance} \leftarrow \text{prevDistance}
\]

\text{loop}

\text{swapDone} \leftarrow \text{true}

\text{for} \ j = 0 \ \text{to} \ \text{deg}(n) \ \text{do}

\text{swap}(n, j - 1, j)

\text{newDistance} \leftarrow \text{updateOrderedTED}(t_1, t_2)

\text{if} \ \text{newDistance} < \text{minDistance} \ \text{then}

\text{minDistance} \leftarrow \text{newDistance}

\text{swapDone} \leftarrow \text{false}

\text{else}

\text{swap}(n, j - 1, j)

\text{end if}

\text{end for}

\text{if} \ \text{swapDone} \ \text{then}

\text{return} \ \text{minDistance}

\text{end if}

\text{end loop}

\text{Figure 4.14:} A function that finds for an order-indifferent node \(n\), the order of children that minimizes \(\text{OTED}(t_1, t_2)\).

\textbf{Reusing Dynamic Data Between Multiple Computations of \textit{OrderedTED}}

We now show how to apply a dynamic programming approach in the search for a local maximum of the similarity. The main principle is to reuse results of existing computed distances between subtrees, i.e., use results of computations from previous iterations, when examining a swap of two children of an order-indifferent node. To illustrate the difficulty, assume that we swap two adjacent children \(c_1\) and \(c_2\) of some node \(n\) of \(t_1\), as depicted in Figure 4.15. The swap changes the postorder numbering of all the nodes in the subtrees rooted at \(c_1\) and \(c_2\), inclusively, however, the postorder numbering of the rest of the nodes remains unchanged. Consequently, the function \(\text{lm-leaf}()\) changes only for all the ancestors of \(n\) and for all the descendents of \(c_1\) and \(c_2\), including \(c_1\) and \(c_2\).

After the change, \(\text{sted}(i, j)\) from the previous computation of \(\text{OTED}\) is still relevant for all indexes \(i\) such that \(i \notin \text{ancestors}(n)\). So, it is sufficient to compute the values of \(\text{sted}(i, j)\) in the original order only for \(i \in \text{ancestors}(n)\). The problem is that the values of \(\text{sted}(i, j)\) are ordered according to the postorder numbering before the swap. Hence, for each offspring \(i\) of \(a\) and \(b\) and for each node \(j\) of \(t_2\), we should update the \(\text{sted}\) table in the following way: \(\text{sted}(i, j) = \text{sted}_{\text{prev}}(\text{new2old}(i), j)\)

The mapping \(\text{new2old}\) is a mapping of the postorder numbering of \(t_1\) after the swap to the postorder before the swap. This mapping can be obtained in a single postorder traversal of the subtrees rooted at \(a\) and \(b\).

If as a result of a swap, the leftmost child of \(n\) is changed from \(a\) to \(b\), as in Figure 4.15, then \(\text{lm-leaf}(i)\) also must be changed for all \(i \in \text{ancestors}(n)\), in the following way:
Figure 4.15: Tree $t_1$ before the swap of $c_1$ and $c_2$

$lm-leaf(i) = lm-leaf_{prev}(b)$.

The update is performed by algorithm $updateOrderedTED$, presented in Figure 4.16. The implicit input of this algorithm comprises the data structures of the previous invocation of $OrderedTED$: $sted_{prev}$, $KeyRoots(t_1)_{prev}$ and $lm-leaf_{prev}()$. This algorithm calculates $OTED(t_1, t_2)$ after the swap of the adjacent children $a$ and $b$, of $n$. 
UpdateOrderedTED\( (n, a, b, t_1, t_2) \)

**Preprocessing:**
create a new2old mapping and update function \( lm\text{-}leaf() \)
for all \( i \) in subtrees rooted at \( a \) and \( b \) do
  for \( j \leftarrow 1 \) to \( |KeyRoots(t_2)| \) do
    \( sted(i, j) \leftarrow sted\text{-}prev(new2old(i), j) \)
  end for
end for

\( KeyRoots(t_1) \leftarrow KeyRoots\text{-}prev(t_1) \cap ancestors(n) \)

**Main loop:**
for \( i' \leftarrow 1 \) to \( |KeyRoots(t_1)| \) do
  for \( j' \leftarrow 1 \) to \( |KeyRoots(t_2)| \) do
    \( i \leftarrow KeyRoots(t_1)[i'] \)
    \( j \leftarrow KeyRoots(t_2)[j'] \)
    Compute \( sted(i, j) \)
  end for
end for

return \( sted(|t_1|, |t_2|) \)

**Figure 4.16:** Update \( OTED(t_1, t_2) \) after a swap of \( a \) and \( b \) that are adjacent children of a node \( n \) of \( t_1 \).

As an example of reusing previous computations, as part of the dynamic programing, consider the two trees depicted in Figure 4.17. Here, the left-to-right postorder ordering is presented, instead of the names of the nodes. The stored intermediate computations of \( OTED \), that is the tree edit distances between all the subtrees, appear in Table 4.2.

![Figure 4.17: The trees before the swap of nodes 3 and 4 of the left tree.](image)

In order to update the distances between all the subtrees, after a swap of the nodes 3 and 4 of the left tree, it is sufficient to recompute the distances between the subtree rooted at Node 5 of the left tree and all the subtrees of the right tree. This is due to the fact that distances between other subtrees are unchanged. This means that the fifth column of the distance table should be updated. Before the update, we also must change the order of the relevant columns, since as a result of the swap, the left-to-right postorder numbering of the nodes is changed.
Table 4.2: The stored intermediate computations of the TED algorithm. Here, $sted(2, 3) = 1$ means that the OTED between the subtree rooted at node 2 of the left tree and the subtree rooted at node 3 of the right tree is 1.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 4.18: The trees after the swap of the nodes of the left tree ordered 3 and 4 before the swap.

Table 4.3: The stored intermediate distances of the TED algorithm, before the main loop of the updateOrderedTED algorithm.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
The first four columns of the table after the swap are the permutation of the first four columns of the table before the swap, according to the change of the postorder numbering of the nodes. The permutation is: 1 ← 4 ← 3 ← 2 ← 1. Figure 4.18 and Table 4.3 present the trees and the distance table after the swap, but before the update of the fifth column. Then according to the code of the main loop of the updateOrderedTED algorithm, the fifth column is updated as follows: sted(5, 1) = 4, sted(5, 2) = 5, sted(5, 3) = 4 and sted(5, 4) = 3. This is the OTED between the trees after the swap. The result of the update shows that the tree edit distance between the trees after the swap is still 3.

4.1.4 Complexity Analysis

In general, the problem of finding the unordered tree edit distance of two trees is NP-hard [29]. SearchTED finds a local optimum of OTED(t1, t2), and in this section we discuss its time and space complexity.

The initial ordered tree edit distance OTED(t1, t2) is less then |t1| + |t2|. This is due to the fact that one can always delete all the |t1| nodes of t1 and insert |t2| nodes of t2. Thus, the distance OTED(t1, t2) can only be reduced at most |t1| + |t2| times, since the distance is never increased during the search. Between two consecutive times when the distance is reduced, the search recomputes the distance at most O(|UC(t1, t2)|) times. Where UC(t1, t2) is the set of all children of unordered nodes of t1 and t2. The time complexity of the distance update is bounded by the time complexity of the original OTED algorithm, presented in Figure 4.7, that is, |t2| × |t1| × min(depth(t1), leaves(t1)) × min(depth(t2), leaves(t2)). The expression min(depth(t1), leaves(t1)) is equal to depth(t1), since syntax trees of SQL queries have very few nodes with a single child. Finally, the worst case time complexity of the search is:

\[ O((|t1| + |t2|) × |t2| × |t1| × depth(t1) × depth(t2) × |UC(t1, t2)|) \]

The space complexity is the same as for the original OTED algorithm, that is the size of the sted table—O(|t2| × |t1|).

Example

Here we present an example of calculating the similarity between two queries using the Tree Edit Distance method. In order to calculate the similarity between the following two queries:

\[
\begin{align*}
\text{SELECT} \ x \ \text{FROM} \ A \ \text{WHERE} \\
&\quad \quad \text{x in (SELECT \ x \ \text{FROM} \ B \ \text{NATURAL JOIN} \ C \ \text{WHERE} \ C.c = 0 \ \text{AND} \ B.x > 100)} \\
&\quad \quad \quad \quad \quad \quad \quad \text{AND} \ y \ \text{in (SELECT} \ y \ \text{FROM} \ D \ \text{JOIN} \ E \ \text{WHERE} \ E.z > D.z) \\
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} \ x \ \text{FROM} \ A \ \text{WHERE} \\
&\quad \quad \text{x in (SELECT} \ x \ \text{FROM} \ B \ \text{NATURAL JOIN} \ C \ \text{WHERE} \ C.c = 0 \ \text{AND} \ B.x > 100) \\
&\quad \quad \quad \quad \quad \quad \quad \text{AND} \ y \ \text{in (SELECT} \ y \ \text{FROM} \ D \ \text{JOIN} \ E \ \text{WHERE} \ E.z > D.z)
\end{align*}
\]
we first extract \( t_1 \) and \( t_2 \)—the syntax trees of the queries:

Here, \( \text{subtree}_2 \) corresponds to \((\text{SELECT } y \text{ FROM } D \text{ JOIN } E \text{ WHERE } E.z > D.z)\)

\( \text{subtree}_1 \) corresponds to

\((\text{SELECT } x \text{ FROM } B \text{ NATURAL JOIN } C \text{ WHERE } C.c = 0 \text{ AND } B.x > 100)\)

and \( \text{subtree}_3 \) corresponds to

\((\text{SELECT } x \text{ FROM } B \text{ NATURAL JOIN } C \text{ WHERE } B.x > 100 \text{ AND } C.c = 0)\)

The syntax trees are isomorphic if we do not account for the subtrees rooted at the node labeled \text{AND} of \( t_1 \) and \text{OR} of \( t_2 \). So let us concentrate on calculating the TED between

The SearchTED algorithm first calculates the OTED between the trees as they are and then searches for the nodes that can be reordered in order to decrease the OTED. The trees has 6 unordered nodes and the function SelectNodeToSort will first try to reorder the nodes with the most number of offspring. That is for example the \text{AND} node.
of \( t_1 \). The only possible reordering of two children of the node results in the following tree:

\[
\text{AND} \\
\text{in} \quad \text{in} \\
\text{y subtree}_2 \quad \text{x subtree}_1
\]

that clearly decreases the TED(\(t_1, t_2\)) since \(\text{subtree}_1\) is very similar to \(\text{subtree}_3\). The next iteration of the SearchTED will try to reorder the OR node of \(t_2\) but this will not improve the TED(\(t_1, t_2\)).

The subtrees \(\text{subtree}_1\) and \(\text{subtree}_3\) are similar and the next iterations of SearchTED will make them identical by changing the order of the AND node of \(\text{subtree}_3\). Finally, the TED(\(t_1, t_2\)) is the ordered tree edit difference between the following two trees:

\[
\text{AND} \\
\text{in} \quad \text{in} \\
\text{x subtree}_1 \quad \text{y subtree}_2 \\
\text{OR} \\
\text{in} \quad \text{in} \\
\text{y subtree}_1 \quad \text{x subtree}_2
\]

that is the cost of label substitution from AND to OR and twice the cost of label substitution from \(x\) to \(y\). That is 3 if we use the uniform cost. This means that the Tree Edit Distance similarity between the original queries is:

\[
sim_{TED}(t_1, t_2) = 1 - \frac{\text{TED}(t_1, t_2)}{|t_1| + |t_2|} = 1 - \frac{3}{38 + 38} = 0.96
\]

### 4.2 Attribute Analysis

We present now a second method for evaluating the similarity between two SQL queries. This method is based on a premise that similar queries refer to similar sets of attributes. We distinguish between attributes that appear in the constraints of the query and attributes that are merely projected in the SELECT clause. Clearly, two queries that have completely different attribute sets, in their constraints, are not likely to be similar, (neither semantically nor syntactically). However, two queries that have different attribute sets, all over the query and not just in the constraints, may still be similar, both semantically and syntactically. So, when comparing queries, this method takes into consideration only the information about the attributes that are part of the queries. We start by providing some definitions and notations.

**Definition 4.2.1** (Projection Attributes). Given a query \(q\), the set of projection attributes of \(q\) is the set consisting of all the attributes that appear explicitly or implicitly in the projection part of \(q\). We denote this set by \(P_q\).

For example, for the following query:
SELECT name, bonus FROM Employees NATURAL JOIN (SELECT id, salary*12 as bonus FROM Salaries)

the set of projection attributes contains the attributes name, bonus and id since they appear explicitly among the projected attributes of the query, and the attribute salary, which is an implicit projection attribute as it is used to calculate the bonus attribute.

Definition 4.2.2 (Constraint Attributes). Given a query $q$, the set of constraint attributes of $q$ comprises all the attributes that appear in the constraints of $q$. We denote the set of constraint attributes of $q$ by $C_q$.

For example, in the following query:

```
SELECT name FROM Employees NATURAL JOIN Salaries
ON salary > 1000 AND dept_id NOT IN (SELECT dept_id FROM Department WHERE departmentName = 'HR')
```

the constraint attributes are salary, departmentName, dept_id, and also id, which is used implicitly for the NATURAL JOIN.

Given two queries $q_1$ and $q_2$, with projection attributes $P_{q_1}$ and $P_{q_2}$ and constraint attributes $C_{q_1}$ and $C_{q_2}$, the Attribute Analysis similarity is defined using Jaccard set distance (recall Definition 3.2.5):

$$
sim_{AA}(q_1, q_2) = \frac{|P_{q_1} \cap P_{q_2}|}{|P_{q_1} \cup P_{q_2}|} \cdot \frac{|C_{q_1} \cap C_{q_2}|}{|C_{q_1} \cup C_{q_2}|}
$$

To illustrate this definition, suppose $q_1$ and $q_2$ are the following queries:

```
SELECT name, age FROM People
WHERE People.age > 30 AND id in (SELECT id FROM Graduate WHERE university = 'Technion')
```

```
SELECT name, age, university FROM People NATURAL JOIN (SELECT id, university FROM Graduate WHERE country = 'Sweden')
```

The constraint-attributes sets and the projection-attributes sets, of the two queries, are:

- $P_{q_1} = \{\text{name, age}\}$;
- $C_{q_1} = \{\text{age, Graduate.id, People.id, university}\}$;
- $P_{q_2} = \{\text{name, age, university}\}$;
- $C_{q_1} = \{\text{Graduate.id, People.id, country}\}$;

The similarity between these two queries is computed in the following way:

$$
sim_{AA}(q_1, q_2) = \frac{|P_{q_1} \cap P_{q_2}|}{|P_{q_1} \cup P_{q_2}|} \cdot \frac{|C_{q_1} \cap C_{q_2}|}{|C_{q_1} \cup C_{q_2}|} = \frac{|\{\text{name, age}\}|}{|\{\text{name, age, university}\}|} \cdot \frac{|\{\text{Graduate.id, People.id}\}|}{|\{\text{Graduate.id, People.id, country, age, university}\}|}
$$
\[ \frac{2}{3} \cdot \frac{2}{5} = \frac{4}{15} \]

4.2.1 Advantages and Drawbacks of the Attribute Analysis Method

The main advantage of the attribute analysis measure is its efficiency. It merely extracts the projection attributes and the constraint attributes of each query, and then, for each pair of queries it calculates the sizes of the intersection and the union of two sets. Another advantage of this method is the small number of false negative errors. This means that a low similarity evaluated by the attribute analysis method can be trusted. Indeed, if two queries are similar, the sets of attributes they use are the same, or almost the same. These two advantages—efficiency and high accuracy in detecting dissimilar queries—make the method useful for comparing queries before running more computationally expensive and more accurate methods. When the attribute analysis method indicates a low similarity for some pair of queries, then we can avoid executing other, possibly more accurate but less efficient, methods. This helps increasing the efficiency of the search process.

The main drawback of the attribute analysis method is the large number of false positive errors. That is, when two queries are deemed similar by this method, they may still be syntactically and semantically dissimilar. Thus, in case of a positive answer, we should use more accurate method to verify the results. This is due to the fact that many queries may have similar attribute sets and still be totally dissimilar. For example the following two queries \( q_1 \) and \( q_2 \):

```
SELECT id FROM People NATURAL JOIN Salaries
    ON salary > 500 AND
    age = (SELECT max(age) FROM People)

SELECT id FROM People
    WHERE age > 30 AND
    id NOT IN (SELECT id FROM Salaries WHERE salary = 500)
```

have the same attribute sets: \( P_{q_1} = P_{q_2} = \{id\} \), \( C_{q_1} = C_{q_2} = \{salary, age, id\} \), so \( sim_{AA}(q_1, q_2) = 1 \), whereas these queries are totally different.

4.2.2 Complexity

The computation of similarity using the attribute-analysis method, between two queries \( q_1 \) and \( q_2 \), includes parsing the queries, extracting the projection and constraint attributes, and finally, calculating the sizes of the intersections and unions of the attribute sets. The parsing and the attribute sets extraction can be done in \( O(|q_1| + |q_2|) \) time, where \( |q| \) is the number of tokens in the query \( q \). The calculation of the sizes of the intersection and union of any two sets can be done in \( O(s \cdot \log(s)) \) time, where \( s \) is the size of the larger set. This can be done by sorting all the sets in a lexicographic order,
and then applying linear time calculation of the intersection and union sizes. So, the
time complexity of calculating attribute analysis similarities of two queries $q_1$ and $q_2$ is
\[ O(|q_1| + |q_2| + (s_1 + s_2) \cdot \log(s_1 + s_2)), \]
where $s_1$ is the size of the larger set among $C_{q_1}$ and $P_{q_1}$, and $s_2$ is the size of the larger set among $C_{q_2}$ and $P_{q_2}$.

To calculate the attribute analysis similarity for each set of $n$ queries $q_1, q_2, \ldots, q_n$, we parse all the queries, extract all the attribute sets and sort the attribute names of each attribute set, in a lexicographic order. The time complexity of this computation is:
\[ O\left(\sum_{i=1}^{n} (|q_i| + s_i \cdot \log(s_i))\right) \]
where $s_i$ is the size of the larger set among $C_{q_i}$ and $P_{q_i}$. Thus, the total time complexity of the computation of the similarities, for any pair of queries in the set, including the preparation phase, is
\[ O\left(\sum_{i=1}^{n} (|q_i| + s_i \cdot \log(s_i)) + \sum_{i=1}^{n} \sum_{j=i+1}^{n} (s_i + s_j)\right) \]
Note that this is due to the fact that the Jaccard distances for all the relevant pairs of sets, can be calculated in time complexity that is linear in the size of the largest set among the sets participating in the computation. This provides an efficient computation, since the only calculation that is performed for each pair of queries is the calculation of intersection and union sizes, while the sizes of attribute sets, for both projection and constraints, are typically small.

4.3 Vector Model

The third query-similarity measure we present in this work is an adaptation of the commonly-used vector model [4]. We represent each query as a vector of terms and use the similarity between vectors to measure similarity between queries. For instance, the following query, which consists of 12 words,

```
SELECT name FROM People WHERE age > 30 AND age < 40
```

uses 11 terms. Except for SQL reserve words and numeric constants, the terms in this query are the table name People, the attribute names name and age and the operations AND, > and <. Each dimension of such term vector represents a different term. In this example, the query will be represented as a vector that contains 10 times the digit 1, for each one of the 10 terms that appear once in the query, and the digit 2 for the term age, which appears twice in the query. The vector has the following form:

\[[00111001100100010000000200000000001000000001000000001000000001]\]
where each zero corresponds to some term that exists in the repository but does not appear in the query. The vector similarity between two queries $q_1$ and $q_2$, whose vectors are $v_{q_1}$ and $v_{q_2}$, is the product of the vectors $v_{q_1}$ and $v_{q_2}$, i.e.,

$$sim_{VM}(q_1, q_2) = \frac{(v_{q_1}, v_{q_2})}{\|v_{q_1}\| \cdot \|v_{q_2}\|}$$

4.3.1 An Improved Vector Model

The vector model is typically used for unstructured texts, because it does not take into account the order of the terms in the text. However, SQL queries are structured, thus, the vector model is not well suited for measuring similarity between SQL queries. In this section, we present an adjustment of the vector model to SQL, trying to take the structure into account, to some extent. To illustrate the problem, consider the following two queries, which are identical from the vector model point of view, since they contain the same terms.

```
SELECT id FROM Salaries WHERE salary > 0 AND stock_options = 0
SELECT id FROM Salaries WHERE salary = 0 AND stock_options > 0
```

Clearly, the two queries have different constraints, so these queries should be deemed dissimilar. In this example, a search based on the vector model does not take into account the relationships between terms. To cope with this problem, we merge related terms in a way that represents the constraints correctly, creating merged terms. In the above example, the merged terms of the first query are ‘salary > 0’ and ‘stock_options = 0’. For the second query, the merged terms are ‘salary = 0’ and ‘stock_options > 0’. Such merge reduces the similarity between the queries.

We merge terms in the following cases: an operator whose operand is a constant, an expression, an attribute name, or a subquery. We add the merged terms to the terms vector. We also merge the SELECT term with the FROM term, creating a single SELECT-FROM merged term, just for counting their appearance once and not twice, since these two terms always appear jointly. We merge NATURAL and JOIN into a single term NATURAL JOIN, and do the same for OUTER JOIN and all the other types of join. For a binary operator, such as $>$, $<$, $=$ and $!=$, we create a new term that contains the operator and its two operands. But the problem is that there are different merged terms that are equivalent, i.e., have the same meaning. For example the merged terms ‘salary > bonus’ and ‘bonus < salary’ are lexically different one from the other, however, they represent the same constraint and should be considered as the same merged term. In order to solve this problem, we sort the operands lexicographically, and replace the operator with its inverse operator, if needed. For example, the following consecutive terms ‘b > a’ should be merged to provide ‘a < b’. Note that the operator ‘$>$’ has been replaced by ‘$<$’, to preserve the semantics of the constraint. If one of the operands is a subquery, then the subquery is replaced by the string ‘(SUBQUERY)’,
in the merged term. For example, ‘... x < (SELECT ...)’ yields the merged term ‘x < (SUBQUERY)’.

### 4.3.2 Using Full Attribute Names

The same attribute name may appear in the schemas of two different tables. This may cause ambiguity. Thus, in such case, it is important to replace the basic (short) form of the attribute name with the full name, as illustrated in the following example:

```sql
SELECT id FROM Graduate
WHERE id NOT IN (SELECT id FROM People)

SELECT id FROM People
WHERE id NOT IN (SELECT id FROM Graduate)
```

These two queries have the same set of terms. They both give rise to the merged term ‘id NOT IN (SUBQUERY)’, if we use the short form of the attribute name id. Changing the term by using the full form of the attribute names will provide different merged terms. The first query will yield the merged term ‘Graduate.id NOT IN (SUBQUERY)’, whereas the second query will provide ‘People.id NOT IN (SUBQUERY)’, which is a different merged term.

Thus, in order to compare the following two queries, \( q_1 \) and \( q_2 \), respectively,

```sql
SELECT id, salary FROM People NATURAL JOIN
(SELECT id, salary FROM Salaries WHERE salary > (SELECT id FROM Managers))

SELECT id, salary FROM Salaries
WHERE id NOT IN (SELECT id FROM Managers NATURAL JOIN Salaries
WHERE salary > 1000)
```

we construct the representing vectors of the queries \( v_{q_1} \) and \( v_{q_2} \). The term vectors are presented in Table 4.4, where only terms with a non-zero value in at least one of the vectors are presented.
Table 4.4: The term vectors $v_{q_1}$ and $v_{q_2}$ of the queries $q_1$ and $q_2$ (limited to terms with a non-zero value in at least one vector).

<table>
<thead>
<tr>
<th>Term</th>
<th>$v_{q_1}$</th>
<th>$v_{q_2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT-FROM</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>WHERE</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>NATURAL JOIN</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>People</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>People.id</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Salaries</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Salaries.id</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Salaries.salary</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Managers</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Managers.id</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Salaries.salary &gt; (SUBQUERY)</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Salaries.id NOT IN (SUBQUERY)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Salaries.salary &gt; 1000</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The product of the vectors provides:

\[
sim_{VM}(q_1, q_2) = \frac{\langle v_{q_1}, v_{q_2} \rangle}{\|v_{q_1}\| \cdot \|v_{q_2}\|} = \frac{\langle v_{q_1}, v_{q_2} \rangle}{\sqrt{\langle v_{q_1}, v_{q_1} \rangle} \cdot \langle v_{q_2}, v_{q_2} \rangle} = \frac{22}{\sqrt{27} \cdot 30} = 0.77
\]

4.3.3 Discussion

The main advantage of a similarity measure based on the vector model is the efficiency of the computation. It only requires parsing the queries, constructing the term vectors and multiplying the vectors. Creating the vectors and multiplying them has $O(n)$ time complexity, where $n$ is the number of terms in the compared queries.

This improved vector model works relatively well for syntactic similarity, however, frequently it provides poor results for semantic similarity, since minor changes in a query may result in significant changes in the semantics of the query, while only slightly changing the vector of terms, or may even not change it at all. In general, this method does not take into account the relationships among terms and their position in the query, thus, its accuracy is low. However, since this method is efficient and can easily detect queries that are significantly dissimilar, we can use it prior to applying the Tree Edit Distance method, when we examine syntactic similarity between queries. When the product of the term vectors is small, then the queries are dissimilar and we can avoid the costly invocation of the TED method.

4.4 Results Analysis

We now present a different approach that evaluates the similarity between queries based on the answers to the queries over different databases. To apply this approach effectively,
we construct synthetic databases and pose the queries over these databases. The main challenge is to create synthetic databases that can precisely indicate similarity or lack of similarity, between compared queries. Thus, our fourth method, namely Result-Analysis Similarity, is of using synthetic database instances to compare queries by their results.

Consider two queries \( q_1 \) and \( q_2 \), a set of database samples \( S = \{ D_1, D_2, \ldots, D_n \} \) and let \( r_{D_1}^1, r_{D_2}^1, \ldots, r_{D_n}^1, r_{D_1}^2, r_{D_2}^2, \ldots, r_{D_n}^2 \) be the results of evaluating \( q_1 \) and \( q_2 \) over samples from \( S \). Let \( sim_{RA}(q_1, q_2, D_i) \) be the similarity between two queries based on a single database sample \( D_i \), as will be defined hereafter. The Result-Analysis Similarity is the harmonic mean of the similarities \( sim_{RA}(q_1, q_2, D_i) \), for \( 1 \leq i \leq n \) such that \( r_{D_1}^1 \cup r_{D_2}^2 \neq \emptyset \), i.e.,

\[
sim_{RA}(q_1, q_2, S) = \frac{|S_{q_1,q_2}|}{\sum_{D \in S_{q_1,q_2}} \frac{1}{sim_{RA}(q_1, q_2, D)}}
\]

where \( S_{q_1,q_2} = \{ D \mid D \in S, r_{D_1}^1 \neq \emptyset \lor r_{D_2}^2 \neq \emptyset \} \) is the subset of \( S \) that consists of those database samples relevant for the pair of queries \( q_1 \) and \( q_2 \).

In the following sections we define \( sim_{RA}(q_1, q_2, D) \)—the similarity between two queries that is calculated using a single database sample, we explain the motivation behind the definition of \( S_{q_1,q_2} \), and we explain how database samples are generated.

### 4.4.1 Similarity Based on a Single Database Sample

Let \( r_{D_1}^1 \) and \( r_{D_2}^2 \) be the results of the queries \( q_1 \) and \( q_2 \) over a database sample \( D \). To compare \( q_1 \) and \( q_2 \), we first create a partial matching between the attributes of \( r_{D_1}^1 \) and the attributes of \( r_{D_2}^2 \), by comparing their projection attributes.

**Definition 4.4.1.** The *projected attributes* of the \( i \)-th column of a query \( q \) is the set of all the attributes that appear explicitly or implicitly (i.e., used for computing the attribute) in the \( i \)-th column of the query results. We denote this set by \( P_q(i) \).

For example, the following query

```
SELECT max(age)-max(year), name, salary*12
FROM People JOIN Salaries
```

has a schema with three attributes, and the sets of projected attributes are: \( P_q(1) = \{ age, year \} \), \( P_q(2) = \{ name \} \), \( P_q(3) = \{ salary \} \).

We match two result attributes of compared queries only if their sets of projected attributes are identical. For example, consider the first attribute in each one of the following three queries. They all have the same set of projected attributes, comprising the column name.

```
SELECT name FROM People
SELECT max(name) FROM People
SELECT name FROM People Where age > 20
```
This means that these three queries can be compared using the Result Analysis approach.

In contrast, consider the following three queries \( q_1, q_2 \) and \( q_3 \):

- \( \text{SELECT age+year FROM People} \)
- \( \text{SELECT name FROM People} \)
- \( \text{SELECT max(age), year FROM People Where year > 1990} \)

The sets of projected attributes are as follows: \( P_{q_1}(1) = \{ \text{age, year} \} \), \( P_{q_2}(1) = \{ \text{name} \} \), \( P_{q_3}(1) = \{ \text{age} \} \) and \( P_{q_3}(2) = \{ \text{year} \} \). Clearly, no pair of \( q_1, q_2 \) and \( q_3 \) can be compared using the Result Analysis method, since the attributes of their answers have different sets of projected attributes.

If there are several possible matches for the sets of projected attributes, of the compared queries, then we examine all the possible matches, and return as the similarity value the maximal similarity among these matches. In practice, there expected to be only a small number of mappings in which sets of projected attributes are equal, so this comparison is feasible. For a given matching, we order the columns according to the matching, i.e., if an attribute \( A_1 \) of the result of \( q_1 \) is matched to attribute \( A_2 \) of the result of \( q_2 \), then they are both the \( i \)-th attribute of \( r^D_1 \) and \( r^D_2 \), for some \( i \). We do not include in \( r^D_1 \) and \( r^D_2 \) the columns that are not part of the matching. We compare the results as multisets of rows:

\[
\text{sim}_{RA}(q_1, q_2, D) = \frac{|r^D_1 \cap r^D_2|}{|r^D_1| + |r^D_2| - |r^D_1 \cap r^D_2|}
\]

We use Jaccard Similarity, as defined in Chapter 3 (Definition 3.2.6). Note that the intersection \( r^D_1 \cap r^D_2 \) is a multiset intersection (Definition 3.2.4, in Chapter 3).

### 4.4.2 Database Samples Generation

In this section we discuss which database samples should be generated and we present the sample generation algorithm. At the end of the section we present an example that illustrates the database sample generation process and the comparison of queries using generated samples.

If at least one of the two compared queries yields a non-empty result on some database sample, we can use this database sample to compare the two queries. However, when the two queries return an empty result on some database sample, then this sample cannot indicate whether the queries are similar or not. Hence, the database sample generation algorithm is built on the premise that for each repository query it is required to generate at least one database sample that produces a non-empty result for the query. This will guarantee that for each pair of queries \( q_1 \) and \( q_2 \), the set of relevant database samples \( S_{q_1,q_2} \) is non-empty, and then any two queries are comparable. The following algorithm creates database samples.
\( S \leftarrow \emptyset \)

\[ \text{for all repository queries } q \text{ do} \]
\[ \quad \text{if there is no } D \in S \text{ s.t. } |\mathcal{R}_q(D)| \neq 0 \text{ then} \]
\[ \quad \quad D \leftarrow \text{generateDatabaseSample}(q) \]
\[ \quad \quad S \leftarrow S \cup \{D\} \]
\[ \text{end if} \]
\[ \text{end for} \]

**Figure 4.19:** Generation of database samples.

The function `generateDatabaseSample(q)` creates a database sample for which the query \( q \) returns a non-empty result. First, it extracts the sets of relevant tables, i.e. the tables used in the query, and the set of relevant attributes. It includes in the set of relevant attributes all the primary keys of the relevant tables, even if these keys do not appear in the queries. Then, for each relevant attribute \( A \), it creates the set of relevant values, denoted \( \mathcal{R}(A) \), by analyzing the constraints of the query. For example, suppose some numeric attribute \( A \) of query \( q \) appears in the following constraints: \( A = 5 \), \( A > 10 \) and \( A < 100 \). Then, the domain of \( A \) (all the values that can be assigned to \( A \)) can be split into four sets such that each set is defined by the satisfaction, or dissatisfaction, of some subset of the three constraints. Table 4.5 presents the four sets.

Table 4.5: Four sets of values for \( A \), w.r.t. the constraints \( A = 5 \), \( A > 10 \) and \( A < 100 \).

<table>
<thead>
<tr>
<th></th>
<th>( A = 5 )</th>
<th>( A &gt; 10 )</th>
<th>( A &lt; 100 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \leq 10, A \neq 5 )</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>( A = 5 )</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>( A &gt; 10, A &lt; 100 )</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>( A \geq 100 )</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

Our goal is to create a database sample on which the query \( q \) yields a non-empty result. Thus, \( \mathcal{R}(A) \) must contain at least one value from each of the four sets that appear in Table 4.5. Otherwise, the effective constraints may not be satisfied. For example, suppose \( \mathcal{R}(A) \) does not contain any value that satisfies \( A \leq 10 \) and \( A \neq 5 \). Then for any database sample that generates values of \( A \) from \( \mathcal{R}(A) \) the following query returns an empty answer.

**SELECT A FROM T WHERE A < 100 AND NOT ( A > 10 AND A = 5)**

We want to guarantee that our method will work for such query by building a proper database sample. Therefore, we add to \( \mathcal{R}(A) \) at least one value from each one of the four sets. For attributes whose domain is not ordinal (no order is defined), there are only two types of constraints: \( A = \text{const} \) and \( A \neq \text{const} \). For such attributes, we add to \( \mathcal{R}(A) \) all the constant that appear in the constraints and a single additional arbitrary value. For example, suppose the attribute `city` appears with the values ‘London’ and ‘Paris’ in constraints of a query \( q \). Then we add ‘London’, ‘Paris’ and, some
arbitrary value, say ‘Haifa’, to $RV(city)$. We add different values to $RV(A)$ until its size exceeds a predefined threshold $minValues$. These values are extracted from constraints of other queries, from real database instances or generated randomly. A large $minValues$ threshold increases the probability to find an appropriate database sample, however, increasing $minValues$ leads to an exponential increase of the worst-case time complexity of the database sample generation.

The algorithm that is presented in Figure 4.20 generates the database sample by inserting rows of attribute values into the relevant tables. Instead of filling the relevant tables one by one, the algorithm randomly selects a set of values for each relevant attribute and inserts a single row generated from these values into each table. After each insertion, it checks if the compared query has a non-empty result over the sample table, and when it does, the generation of the database sample is terminated.

We refer to an assignment of a value to each relevant attribute as an attribute-assignment vector.

**Definition 4.4.2.** Let $A_1, A_2, \ldots, A_n$ be all the relevant attributes of a query $q$ and let $RV(A_1), RV(A_n), \ldots, RV(A_n)$ be the sets of selected values for each attribute. Then, $x \in RV(A_1) \times RV(A_2) \times \ldots \times RV(A_n)$ is an attribute-assignment vector of $q$. Given a table $t$, we denote by $x|t$ the vector that is obtained from vector $x$ by omitting the values of the attributes that do not appear in table $t$.

To generate a database sample for a query $q$, we examine all the attribute-assignment vectors, i.e., all the vectors in $RV(A_1) \times RV(A_2) \times \ldots \times RV(A_n)$. There are $\prod_{i=1}^{n} |RV(A_i)|$ attribute-assignment vectors in this set, however, the number $n$ of relevant attributes, and the number of relevant tables are typically small.

```
function generateDatabaseSample(q)
    Preprocessing:
    extractRelevantValues(q, minValues)
    for all table $t$ in $D$ do
        $t \leftarrow \emptyset$
    end for
    for all $x \in RV(A_1) \times RV(A_2) \times \ldots \times RV(A_n)$ do
        for each relevant table $t$ do
            insert $x|t$ into $t$ with probability $p_t$
            if $|r^D_q| \neq 0$ then
                return $D$
            end if
        end for
    end for
end function
```

**Figure 4.20:** Generating a database sample on which $q$ has a non-empty answer.

The probability $p_t$ can be set locally, for each table, or globally for an entire database. the default value is 0.5 for each table, and if the algorithm fails to find a database
sample for which the query $q$ has a non-empty answer, we increase these probabilities for each table and repeat the algorithm, until it generates the required sample. The default initial value of the parameter $\text{minValues}$ is set to 2, in our implementation, and it is also increased if the algorithm fails to generate the required database sample. In our experiments we never had to increase this parameter more than once.

To illustrate the approach, consider the following query.

\[
\text{SELECT name, id, year, age FROM People NATURAL JOIN Graduate}
\]
\[
\text{WHERE Graduate.year < 2012 AND Graduate.university = 'Technion' AND People.name = 'Alice'}
\]

To generate the database sample for this query, we first extract relevant tables and attributes. The relevant tables are \text{People} and \text{Graduate}. Assuming that the table \text{People} has a numeric key \text{id} that is also an attribute of the \text{Graduate} table, we add this attribute to the list of relevant attributes. So, the relevant attributes are \text{year}, \text{university}, \text{id} and \text{name}. It is important to note that the attribute \text{age} that appears in the query is irrelevant for the sample creation, since it does not appear in the constraints of the query and it is not a key attribute. The attribute \text{year} is compared with 2012, thus, we add to $RV(\text{year})$ two values. One value, say 2011, satisfies the constraint $\text{year} < 2012$ and another value, say 2013, does not satisfy the constraint. The attribute \text{name} is compared with the value ‘Alice’, so we add ‘Alice’ and another string different from ‘Alice’, for example ‘Bob’ to $RV(\text{name})$. Similarly, the column \text{university} is compared with the value ‘Technion’, so we add ‘Technion’ and another string different from ‘Technion’, for example ‘Cornell’ to $RV(\text{university})$. Finally, the attribute \text{id} does not appear in any constraint and is not associated with any constant, in the query, but since it is a key attribute of the \text{People} table, we add two values 1 and 2 to $RV(\text{id})$, where 1 and 2 are \text{id} values that we extract from the real database.

Table 4.6, presents the sets of values chosen for the relevant attributes of $q$.

<table>
<thead>
<tr>
<th>$RV(\text{year})$</th>
<th>2011, 2013</th>
</tr>
</thead>
<tbody>
<tr>
<td>$RV(\text{university})$</td>
<td>Technion, Cornell</td>
</tr>
<tr>
<td>$RV(\text{name})$</td>
<td>Alice, Bob</td>
</tr>
<tr>
<td>$RV(\text{id})$</td>
<td>1, 2</td>
</tr>
</tbody>
</table>

Suppose in the first iteration of the algorithm of Figure 4.20, the assignment vectors is $<2013, \text{Technion}, \text{Alice}, 1>$. The first iteration adds the row $<1, \text{Alice}>$ to the \text{People} table and adds the row $<1, \text{Technion}, 2013>$ to the \text{Graduate} table. At the end of the first iteration, the query $q$ has an empty answer over the sample table, thus, the algorithms proceeds to the second iteration. Suppose that in the second iteration, the assignment vectors is $<2011, \text{Technion}, \text{Alice}, 1>$. At this iteration the algorithm
does not insert the row <1, Alice> to the People table since the attribute id is a primary key attribute of the table. Thus, the algorithm just adds the row <1, Technion, 2011> to the Graduate table. At the end of the second iteration, the query q has a non-empty answer over the sample table, so the algorithms terminates. The generated database are presented in the Table 4.7.

<table>
<thead>
<tr>
<th>People</th>
<th>Graduate</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>university</td>
</tr>
<tr>
<td>1</td>
<td>Technion</td>
</tr>
<tr>
<td>Alice</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.7: Random DB Instance D

Suppose now we want to evaluate the similarity between the queries q1 and q2:

select name, id, year, age from People natural join Graduate
where Graduate.year < 2012 and
    Graduate.university = 'Technion' and
    name = 'Tony'

select name, id, university from People natural join Graduate
where Graduate.year > 2003
    and Graduate.university = 'Technion'
    and name != 'Carol'

We can execute the queries over the above database sample D, to get the results rD1 and rD2, presented in Table 4.8.

<table>
<thead>
<tr>
<th>rD1</th>
<th>rD2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice 1 2011 35</td>
<td>Alice 1 Technion</td>
</tr>
</tbody>
</table>

Table 4.8: The results of queries q1 and q2 on the database sample D.

In order to compare the results we first match their columns. We use the fact that the columns of rD1 are produced from the attributes name, id and university and the columns of rD2 are produced from name, id and year. Consequently the first and the second column of the results are matched, respectively, and the other columns remain unmatched. The similarity between the results is calculated using the following formula:

\[ \text{sim}_{RA}(q_1, q_2, D) = \frac{|r_D^D \cap r_D^D|}{|r_D^D| + |r_D^D| - |r_D^D \cap r_D^D|} = \frac{1}{1 + 2 - 1} = 0.5 \]
4.4.3 Discussion

In the Result Analysis method, in order to find the repository queries that are semantically similar to a given user query, we construct database samples, run the query on a set of database samples and compare the answer of the user query with the answers of the repository queries, over the database sample, using Jaccard similarity to compare the query answers. The database samples, as well as the results of repository queries, are generated offline, in a preprocessing step. The only computation that is done in realtime, for each pair of compared queries, is the evaluation of the queries over the sample databases (which are typically very small) and the comparison of non-empty results using the Jaccard-similarity formula. The time complexity of the comparison between two query answers is $O(n \log n)$, where $n$ is the size of the larger answer, since we sort the rows of each answer lexicographically, and then the computation of the multiset intersection requires linear time in the size of the set. This method was developed for measuring semantic similarity and it is much more accurate than other methods, in detecting semantic similarity, according to our experimental evaluation.
Chapter 5

Query Recommendation System

In this chapter we present our query recommendation system that provides direct and guided search capabilities for repository queries. In a direct search, the user provides a query and the system finds syntactically or semantically similar repository queries, for the query specified by the user. In a guided search, the user does not have to provide her own query. In a syntactic guided search, the user specifies relevant attributes and fragments of the queries. The system finds repository queries that refer to the specified attributes and contain the specified fragments. In a semantic guided search, the user specifies relevant attributes and parts of the results that a relevant query should return, and the system finds repository queries whose answers comply with these requirements.

5.1 Direct Search

In this section we elaborate on direct search, where the user provides her own query, and the system finds repository queries that are semantically or syntactically similar to the query provided by the user. The direct syntactic search is helpful, for example, when a user is not familiar with the database schema but is able to compose a simple, but not necessarily correct, query, as part of the attempt to find and use similar queries that were written by experts. The direct syntactic search can be used to check the existence of some queries in the repository. For example, homework graders in a university course can use the direct syntactic search to find suspiciously similar queries in homework submissions of different students. Bank security systems can use the direct syntactic search to track people who run queries that seem suspicious. For example, a security officer can compose a query $q$ that finds bank accounts with no recorded activity for a long period of time. Then, the officer can use the direct syntactic search to find queries similar to $q$ among the queries posed by the bank employees, and deem these queries as suspicious, if posed by employees that are not suppose to pose such query. Actually, any user of a database system can use the direct search after creating a view, to avoid duplications of views in the system. In many cases, the direct semantic search can be used to find existing queries that were written for the same task as the user query. The
user may decide to use an existing query instead of her query, if the existing query was written by an expert and is more efficient and more precise than the initial query.

In a direct syntactic search, whose schema is presented in Figure 5.1, given a user query, the system creates a syntax tree, the vector of terms and the relevant attribute sets of the user query. We use the parser of SQLite [23] in order to create the syntax tree of the user query. The relevant attribute sets are $P_q$ and $C_q$, which were defined in Definition 4.2.1 and Definition 4.2.2 of Section 4.2. The search system uses the relevant attribute sets and the terms vector to filter out repository queries that are not syntactically similar to the user query according to Attribute Analysis and Vector Model methods. Repository queries that are not filtered out, are compared to the user query, using the Tree Edit Distance method, and the most similar queries are then presented to the user.

In the direct semantic search, whose schema is depicted in Figure 5.2, given a user query, the system evaluates the query on database samples that were previously created, offline, and uses the results to compare the repository queries to the user query. Those queries that are the most semantically similar to the user query are returned to the user. To speedup the computation, the system finds the database samples on which the user query returned a non-empty answer. Only repository queries that have a non-empty answer on these database samples are compared with the user query.

### 5.2 Guided Syntactic Search

We elaborate now on guided syntactic search. In guided syntactic search, the user does not have to provide an initial query. The user selects the set of relevant attributes and
relevant chunks of queries, including query expressions, sub-expressions, GROUP BY clauses, ORDER BY clauses and aggregate functions. Next, we explain more precisely what are chunks.

**Definition 5.2.1.** A semantically-complete subtree of a query syntax tree is a subtree of the parse tree of a query where the root of the subquery satisfies the following. The root is a node that is labeled with (1) a logical operator such as AND or OR, (2) a boolean operator such as > or =, (3) reserved words such as GROUP BY or ORDER BY, or (4) an aggregate function such as sum or avg.

Nodes that represent sub-queries in the semantically-complete subtrees are substituted with a singleton node labeled (SUBQUERY), i.e., by a node that has no children. We present semantically-complete subtrees to the user as strings that we call chunks.

The reason for substituting subtrees that represent subqueries with a node labeled (SUBQUERY) is that we want the chunk to be short enough to be readable by the user. To illustrate this, consider the following query.

```
SELECT pid, sum(points) FROM Memberships WHERE points > 0 AND points < 200 AND pid NOT IN (SELECT pid FROM Forbidden) GROUP BY pid ORDER BY pid
```

Such query yields the following syntactically-complete subtrees, which can be presented as chunks to the user:

```
points > 0
points < 200
pid IN (SUBQUERY)
```
The search process, whose schema is depicted in Figure 5.3, goes as follows.

1. The search starts by letting the user specify relevant attributes by selecting attributes from a list of attributes.

2. The user selects relevant query chunks that are suggested by the system based on the attribute selection.

3. Every time the user adds a chunk, the queries are sorted according to the number of user-selected chunks they contain. In order to simplify the presentation of the most relevant queries, syntactically similar queries are presented as a single cluster and the relevance of a cluster is defined by the most relevant query of the cluster.

4. Finally, at the end of the search, the user may see all the queries of the most relevant clusters.

5.2.1 An Example of a Guided Syntactic Search

Consider a database containing four tables, with the following schemas: Person \((pid, name, firstname, lastname, address)\); Company \((cid, name, address)\); Membership

\[
\begin{align*}
\text{NOT } & \text{pid IN (SUBQUERY)} \\
& (\text{points } > 0) \text{ AND (points } < 200) \\
& (\text{(points } > 0) \text{ AND (points } < 200)) \text{ AND pid NOT IN (SUBQUERY)} \\
\text{ORDER BY } & \text{pid} \\
\text{GROUP BY } & \text{pid} \\
\text{sum(points)} \\
\end{align*}
\]

\textbf{Figure 5.3:} Guided Syntactic Search.
(pid, cid, invitedby, points); and Forbidden (cid, name, address). Suppose a user is searching for a query that finds the identifiers of people who gained between 0 and 200 points.

The guided-search process is as follows. Initially, the user selects the following relevant attributes: pid and points from the list of all attributes. The system presents to the user the chunks of the repository queries that include the selected attributes. The suggested chunks appear in Figure 5.4. Suppose the user selects the chunk points > 0, since it is obviously related to the required task.

Then, the system suggests more-complicated chunks, in addition to previously suggested chunks. For example, the system suggests the chunk points > 0 AND points < 200, since there is a repository query that contains such constraint. Figure 5.5 contains the suggested chunks after the selection of the chunk points > 0 by the user.

After the selection of the points > 0 AND points < 200 chunk, the user can see the clusters with the maximal number of matching chunks and infer there is only one cluster that contains queries relevant to his search. Figure 5.6 shows how the cluster is presented. The cluster is represented by the query that contains the highest number of the user-selected chunks.
Finally, the user may see all the queries of the relevant clusters (in this example it is only one cluster) ordered by their relevance to the search, that is, ordered according to the number of user-selected chunks in the representative query. So far, the user had selected only two clusters: `points > 0 AND points < 200` and `points > 0`. Figure 5.7 presents the queries of the relevant cluster sorted by their relevance to the search.

![Figure 5.6: A cluster that contains 9 queries and at least one of the queries contains 2 chunks selected by the user.](image)

![Figure 5.7: All the queries of the most-relevant cluster, sorted according to the number of user-selected chunks in them.](image)

### 5.2.2 Clustering Similar Queries

When the system contain thousands of repository queries, there may be situations where many queries are equally relevant, by having the same number of user-selected chunks. In such cases, the user will have to look through many queries to find one that satisfies the search task. To facilitate query selection for the user, we apply hierarchical clustering on the queries, where the distance between two queries is the Tree Edit Distance between their syntax trees. The clustering is performed only once, as a preprocessing, before the
user starts the search. When relevant chunks are added, the relevance of the clusters changes and this affects their order. Recall that the relevance of a cluster is determined by the maximal number of chunks in the query. Thus, we use a strict-partitioning clustering, where each query belongs to exactly one cluster. Otherwise, there could be multiple clusters with the same relevance and the same representative query. Such clusters will be indistinguishable, from the user perspective, and this can be confusing for users.

We use a single-linkage clustering that starts by putting each query in a cluster of its own. The clusters are then sequentially combined into larger clusters, until all the queries end up being in the same cluster. The distance between two clusters is defined to be the minimal distance between any pair of a query from one cluster and a query from the other cluster. In each step, the two clusters with the smallest distance between them are merged.

We use the single-linkage clustering algorithm presented in [22]. This clustering algorithm has $O(n^2)$ time complexity, where $n$ is the number of queries. The result of this algorithm is up to $n - 1$ successive partitions of queries where every partition can be created from the previous partition by merging two closest clusters. The first partition contains exactly $n$ clusters and the last partition contains a single cluster. This hierarchy of partitions is later used during the search process in the following way. Every time the user adds a chunk to the list of relevant chunks, we compute the relevance of each cluster, for all the partitions. The time complexity of cluster relevance update is $O(n)$. Then, we decide which partition is the most effective for the user. We assume that a typical user will not look through more than 3 queries, but this number is configurable and can be changed. Hence, we choose the partition whose 3 most relevant clusters provide the highest relevance, in comparison to all the other partitions.

For example, suppose the only relevant chunk the user has selected is points > 0 and there are exactly three queries $q_1, q_2$ and $q_3$ that contain this chunk. Then, we choose a partition in which one cluster contains $q_1$, one cluster contains $q_2$ and one cluster contains $q_3$. If there are several such partitions, the one with the minimal number of clusters is chosen.

Figure 5.8: Hierarchical clustering of the queries $q_1, q_2, q_3, q_4, q_5, q_6$. 

![Hierarchical Clustering Diagram](image-url)
Table 5.1: Possible partitions generated by the hierarchy of Figure 5.8.

<table>
<thead>
<tr>
<th>Partition size</th>
<th>Clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>$q_1, q_2, q_3, q_4, q_5, q_6$</td>
</tr>
<tr>
<td>5</td>
<td>$c_7, q_1, q_2, q_5, q_6$</td>
</tr>
<tr>
<td>4</td>
<td>$c_7, c_8, q_5, q_6$</td>
</tr>
<tr>
<td>3</td>
<td>$c_9, q_5, q_6$</td>
</tr>
<tr>
<td>2</td>
<td>$c_{10}, q_6$</td>
</tr>
<tr>
<td>1</td>
<td>$c_{11}$</td>
</tr>
</tbody>
</table>

Figure 5.8 presents the hierarchy of 6 queries $q_1, q_2, q_3, q_4, q_5$ and $q_6$, and sub clusters $c_7, c_8, c_9, c_{10}$ and $c_{11}$. The number between the parenthesis beside the name of the query or the cluster is the relevance to the search. It is 0 when no query in the cluster contains the chunk $\text{points} > 0$, and 1 otherwise. The system will use Partition 4 to present to the user $q_6, c_7$ and $c_8$ because this is the partition with the smallest number of clusters in which there are at least 3 relevant clusters.

5.2.3 Relevant-Chunk Suggestion

Sometime, some chunks are substrings of other chunks. For example, the chunk $\text{points} > 0$ is a substring of the chunks $(\text{points} > 0) \text{ AND } (\text{points} < 200)$, and a substring of $((\text{points} > 0) \text{ AND } (\text{points} < 200)) \text{ AND pid NOT IN (SUBQUERY)}$.

When the user chooses the $\text{point} > 0$ chunk as the guidance for the search, it is beneficial to show the user all the chunks that contain the chosen chunk. However, typically there would be too many such chunks. Thus, we show only the chunks that are immediate extensions of the chunks that the user has already selected. In such case, if the user has only selected the chunk $\text{point} > 0$, the system will suggest the chunk $(\text{points} > 0) \text{ AND } (\text{points} < 200)$ and will not suggest the chunk $(\text{points} > 0) \text{ AND } (\text{points} < 200) \text{ AND pid NOT IN (SUBQUERY)}$, since the second chunk is not an immediate extension of $\text{points} > 0$.

**Definition 5.2.2.** A chunk $C_x$ is an *immediate extension* of a chunk $C_y$ if and only if $C_y$ is a substring of $C_x$ and there is no chunk $C_z$ such that $C_y$ is a substring of $C_z$ and $C_z$ is a substring of $C_x$.

The system presents to the user all the immediate extensions of the selected chunks. It also presents the shortest chunks that are relevant to the selected attributes. The sets of immediate extensions for each chunk are created offline.

5.3 Guided Semantic Search

In this section we present the guided semantic search. Like in the guided syntactic search, the user does not have to provide a query and the search starts by letting the
user specify the attributes that are relevant for the search. Then, instead of selecting relevant chunks, the user specifies results that relevant queries should return on the database samples suggested by the system.

The schema of the search process is depicted in Figure 5.9. Initially, the user specifies relevant attributes. Then, the system selects and presents one of the precomputed database samples and possible results of repository queries over this sample. The user can select one of the presented results or indicate that a result she expects to be returned by a relevant query is missing. If the user selects one of the provided results, the system presents the corresponding queries, if there are only a few of them, or continues the search process, to reduce the number of queries in the answer, by presenting another database sample and optional results. The system selects database samples in a way that minimizes the number of samples the user should examine, by presenting to the user, in each step, the most selective sample. We elaborate on the selection of the database sample in Section 5.3.2. When the number of repository queries that match the search is small enough, the system presents the queries to the user.

5.3.1 An Example of a Guided Semantic Search

Consider a search over the database whose table schemas are presented at the beginning of Section 5.2.1. Suppose the user wants to find a query that returns the identifiers of people who obtained between 0 and 200 points. First, the user selects relevant attributes

![Figure 5.9: Guided Semantic Search.](image-url)
for the search. Suppose the selected attributes are \texttt{pid} and \texttt{points}. Then, the system selects and presents the database sample depicted in Figure 5.10. The system also presents possible results that the repository queries returned on this database sample, as illustrated in Figure 5.11.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|l|}
\hline
\textbf{THE SAMPLE DATABASE} & & \\
\hline
\textbf{TABLE: person} & & \\
\hline
\textbf{TABLE: company} & & \\
\hline
\textbf{TABLE: memberships} & & \\
\hline
\textbf{name} & \texttt{pid} & \texttt{cid} & \texttt{invitedby} & \texttt{points} \\
\hline
Row 0: & 2 & 920001 & 4 & 100 \\
Row 1: & 4 & 920001 & 5 & 230 \\
Row 2: & 3 & 920001 & 2 & 200 \\
\hline
\textbf{TABLE: forbidden} & & \\
\hline
\end{tabular}
\caption{Database sample presented after selecting the \texttt{pid} and \texttt{points} attributes.}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
\textbf{Result 0 :} \\
\textbf{name: pid} \\
Row 0: & 4 \\
Row 1: & 3 \\
\hline
\textbf{Result 1 :} \\
\textbf{name: pid} \\
Row 0: & 2 \\
\hline
\textbf{Result 2 :} \\
\textbf{EMPTY} \\
\hline
\textbf{Result 3 :} \\
\textbf{name: pid} \\
Row 0: & 2 \\
Row 1: & 4 \\
Row 2: & 3 \\
\hline
\end{tabular}
\caption{The results that repository queries returned on the database sample of Figure 5.10}
\end{table}

In this case, the relevant result is the one in which \texttt{pid} is merely 2, since this is the \texttt{pid} of the only person that has more than 0 and less than 200 points. Thus, the user selects this result the system shows two repository queries that return the result selected by the user.
5.3.2 Database Samples Selection

Each step of a guided semantic search reduces the number of potentially-relevant repository queries. In this section we explain how we select the databases sample that is presented to the user.

In each step, the user examines the suggested database sample and selects one of the suggested result, or she may indicate that no result satisfies the search requirements. Each result selection reduces the number of potentially-relevant repository queries. Let us denote by \( n \) the number of relevant repository queries before the current step, and let us denote by \( s \) the number of relevant queries after the step. The database sample selection should minimize the number of search steps, since each step requires from the user looking at a database sample and potential results, which can be cumbersome.

Each database sample partitions the set of the potentially-relevant repository queries into sets of relevant and irrelevant queries. Obviously, if all the queries have the same answer on the database sample, then this sample should not be presented to the user. Moreover, we select the database sample that minimizes the expected size of \( s \), assuming that any query in the repository has the same probability of \( \frac{1}{n} \) to be the most relevant query. Given some database sample \( D \), let \( s_1, s_2, \ldots, s_k \) be the sizes of the sets of repository queries that are not distinguishable by \( D \). Then the expected number of relevant queries after the current step is:

\[
E[s] = \sum_{i=1}^{k} P[s = s_i] \cdot s_i = \sum_{i=1}^{k} \frac{s_i}{n} \cdot s_i = \frac{1}{n} \cdot \sum_{i=1}^{k} s_i^2
\]

This means that in order to minimize the expected number of relevant repository queries after each step, we should select the database that minimizes the following expression: \( \sum_{i=1}^{k} s_i^2 \). The numbers \( s_1, s_2, \ldots, s_k \) can be calculated in \( O(n^2) \) time complexity, by finding the sizes of connected components in a graph whose nodes are the queries and there is an edge between two nodes if the queries that they represent are indistinguishable by the database sample. Finally, the optimal database sample can be selected with time complexity \( O(n^2 \cdot |S|) \), where \( S \) is the set of all the database samples, by calculating \( \sum_{i=1}^{k} s_i^2 \) for each database sample.
Chapter 6

Experimental Evaluation

In this chapter we present our experimental evaluation. The goal of the experimental evaluation was to compare the accuracy and the performances of the four methods we developed, for measuring semantic and syntactic similarities between queries. We first provide a description of the experiments we performed and the data that were collected to execute the experiments. Then, we present the results of the tests and discuss our conclusions from the experimental results.

6.1 Experimental Setting

To test the methods, we conducted the following type of test. Given a user query, for each method we used the method to sort all the repository queries according to their similarity to the user query. We took the top $k$ queries to be the queries that each method considers to be similar to the user query. In cases where we know for each user query what is the set of repository queries that are actually similar to the user query, we computed the recall and the precision of the method, for different values of $k$.

Recall, precision and F-measure are standard information retrieval quality measures that were presented in Definition 3.2.7 in Chapter 3. We tested different queries as the user query, and for each $k$, we calculated the average recall, precision and F-measure of the results obtained for the tested user queries.

To illustrate our testing methodology, consider as an example, a repository of queries that comprises the 9 queries $q_1, q_2, \ldots, q_9$. Given a user query $q_u$, the repository queries are sorted by their syntactic similarity to the user query, according to each of the methods. The sorted queries are presented in Table 6.1. Assume that the queries that are actually syntactically similar to the user query $q_u$ are $q_1$, $q_2$ and $q_3$. For each value of $k$, in the range 1 to 5, we calculate the recall, precision and F-measure. The results are presented in the Table 6.2.

We performed each experiment multiple times, with different user queries and we calculate the average values of the recall, precision and F-measure for each value of $k$ for the different methods. We tested the different quality measures, for the different
Table 6.1: Repository queries sorted by their syntactic similarity to the query $q_a$, evaluated by the different methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Repository queries, sorted by similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TED</td>
<td>$q_1$ $q_2$ $q_3$ $q_6$ $q_8$ $q_4$ $q_7$ $q_5$ $q_9$</td>
</tr>
<tr>
<td>AA</td>
<td>$q_4$ $q_2$ $q_1$ $q_6$ $q_9$ $q_3$ $q_5$ $q_7$ $q_8$</td>
</tr>
<tr>
<td>VM</td>
<td>$q_1$ $q_2$ $q_3$ $q_8$ $q_6$ $q_7$ $q_5$ $q_9$ $q_4$</td>
</tr>
<tr>
<td>RA</td>
<td>$q_9$ $q_3$ $q_7$ $q_6$ $q_8$ $q_4$ $q_1$ $q_5$ $q_2$</td>
</tr>
</tbody>
</table>

Table 6.2: Recall, precision and F-measure of each method as a function of $k$.

<table>
<thead>
<tr>
<th>Recall</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>TED</td>
<td>0.33</td>
<td>0.66</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AA</td>
<td>0</td>
<td>0.33</td>
<td>0.66</td>
<td>0.66</td>
<td>0.66</td>
</tr>
<tr>
<td>VM</td>
<td>0.33</td>
<td>0.66</td>
<td>0.66</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RA</td>
<td>0</td>
<td>0.33</td>
<td>0.33</td>
<td>0.33</td>
<td>0.33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Precision</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>TED</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.75</td>
<td>0.75</td>
</tr>
<tr>
<td>AA</td>
<td>0</td>
<td>0.5</td>
<td>0.66</td>
<td>0.5</td>
<td>0.4</td>
</tr>
<tr>
<td>VM</td>
<td>1</td>
<td>1</td>
<td>0.66</td>
<td>0.75</td>
<td>0.6</td>
</tr>
<tr>
<td>RA</td>
<td>0</td>
<td>0.5</td>
<td>0.33</td>
<td>0.25</td>
<td>0.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F-measure</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>TED</td>
<td>0.5</td>
<td>0.8</td>
<td>1</td>
<td>0.85</td>
<td>0.85</td>
</tr>
<tr>
<td>AA</td>
<td>0</td>
<td>0.4</td>
<td>0.66</td>
<td>0.57</td>
<td>0.5</td>
</tr>
<tr>
<td>VM</td>
<td>0.5</td>
<td>0.8</td>
<td>0.66</td>
<td>0.85</td>
<td>0.75</td>
</tr>
<tr>
<td>RA</td>
<td>0</td>
<td>0.4</td>
<td>0.33</td>
<td>0.29</td>
<td>0.25</td>
</tr>
</tbody>
</table>

methods, as function of $k$, to examine the effect of $k$ on the performances.

In our experiments, in addition to the four methods that we have developed, we included a fifth method that represent the maximal possible quality. We refer to this method as **Perfect Method**. In this method, the actual similarity of each pair of queries is known and based on this knowledge, the method tries to return the top $k$ queries that maximize recall, precision or F-measure, according to the test. Adding this method allows us to know the proximity of our methods to the best result a method can achieve. Obviously, in real systems, the perfect method cannot be applied because it requires full knowledge of the similarity between queries, and such knowledge does not exist in real searches, but we had it in our experiments.

### 6.2 Datasets

In this section we describe the data we have used in the experiments and how we collected them. In order to be able to measure the recall and the precision, one must assume that for each pair of repository queries it is known if they are similar or not, for the tested type of similarity. We solved this problem in different ways for semantic similarity and syntactic similarity. For semantic similarity we asked people, who participated in the experiment, to write SQL queries that solve different tasks. The user study was
conducted with the help of 53 participants. They all have background in computer science and familiarity with relational databases and SQL. There were 10 tasks for which SQL queries were written. One of the tasks, for example, was to find the employee, in a database of employees of different companies, whose salary is the third from the top in some company, i.e., who receives the third highest salary in the company whose pid is given.

Each participant provided between 5 to 10 different queries. Overall, we collected 400 queries for those 10 tasks. Note that we know for each pair of queries if they were written for the same task, and consequently if they are semantically similar according to Definition 3.1.3.

For syntactic similarity we performed a different experiment. We used the same set of queries that was collected in the previous experiment and asked other participants to mark the pairs of similar queries. Thus for each pair of queries we know if they look similar and consequently are considered syntactically similar according to Definition 3.1.1.

### 6.3 Experimental Results and Discussion

In this section we present the results of measuring recall, precision and F-measure for different $k$ values.

![Figure 6.1: Precision as a function of $k$, for semantic similarity.](image1)

![Figure 6.2: Recall as a function of $k$, for semantic similarity.](image2)
Figures 6.1 - 6.3 present the quality of the methods when measuring semantic similarity. The Result Analysis method provides much better results than the other methods, for any value of \( k \). Note that we have used tasks that deal with similar sets of attributes, thus, the quality of the Attribute Analysis method is very low. Moreover, the Result Analysis method has accuracy that is very close to that of the perfect method. Therefore, it is well-adapted to deal with syntactic search.

Figure 6.3: F-measure as a function of \( k \), for semantic similarity.

Figure 6.4: Precision as a function of \( k \) for syntactic similarity.
Figures 6.4 - 6.6 present the quality of the methods when measuring syntactic similarity. The Tree Edit Distance method provides significantly better quality than the Vector Model and than the Attribute Analysis methods for any value of \( k \). Moreover, the quality of the Tree Edit Distance method is very close to the quality of the perfect method. Thus, this method is well adapted to measure the syntactic similarity of queries in the query recommendation system.

Table 6.3: Average running times, in milliseconds, of a comparison between two queries. The experiments were performed on Pentium(R) Dual-Core CPU T4300 @ 2.10 GHz, Memory 2.00GB, Windows 7 64 bit. The Algorithms were implemented in C.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute Analysis</td>
<td>0.013</td>
</tr>
<tr>
<td>Vector Model</td>
<td>0.51</td>
</tr>
<tr>
<td>Result Analysis</td>
<td>0.9</td>
</tr>
<tr>
<td>Tree Edit Distance</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 6.3 presents the average time of a comparison between two queries, for each method. Note that the Tree Edit Distance method is computationally expensive. That
is the reason why when we measure the syntactic similarity between queries, first we employ the fast Attribute Analysis method, to filter out the pairs of queries that deal with different sets of attributes. Then we use the Vector Model method to filter out pairs that use totally different terms. Only after these two filtering phases, we use the Tree Edit Distance method to evaluate precisely the syntactic similarity. Attribute Analysis method will filter out the methods that deal with different sets of attributes and do not have the potential to be similar and the Vector Model method will filter out queries that use different sets of words and consequently cannot be similar.
Chapter 7

Conclusions

In this thesis, we studied the problem of searching SQL queries in a repository of queries. We explained how such search can be used in recommendation systems, to facilitate and improve the task of writing queries, and we explained how it can be used for monitoring a system, for security purposes. We defined two types of search: syntactic search and semantic search. For these two types of search, we distinguished between two types of similarity between SQL queries—syntactic similarity and semantic similarity. We developed the Tree Edit Distance method for measuring syntactic similarity and the Result Analysis method for measuring semantic similarity. We also adapted to these types of search two additional less-accurate but more-efficient methods, namely Attribute Analysis and Vector Model. We use these methods to reduce the number of queries that should be compared by the more accurate and more computationally-expensive methods, to expedite the search. We compared the accuracy of the adapted methods with the accuracy of the methods that we developed. Our experiments show that the accuracy of the Tree Edit Distance method is significantly better than the accuracy of other methods for measuring syntactic similarity. The accuracy of the Result Analysis method is significantly better than the accuracy of the other methods for measuring semantic similarity.

We used the new search methods to build a query-recommendation system that supports four types of search. In a direct syntactic or semantic search, the system finds syntactically or semantically similar queries to a given user query. In a guided syntactic search, the user selects relevant attributes and syntactic fragments of existing queries and the search system finds queries that include the selected attributes and contain the selected syntactic fragments. In a guided semantic search, the user selects relevant attributes and indicates the results of queries on precomputed synthetic database samples. Future work includes the development of indexes and of other data structures to improve the efficiency and the scalability of the search.
Bibliography


We developed a new method of analyzing the results of SQL queries (Result Analysis) in order to estimate the semantic similarity between SQL queries. In this method, in order to find queries similar to a given user query, we run the query on several database instances and compare the results of the -multisets of rows using the Jaccard metric. Database instances and query results are generated in advance, and the only calculation performed during the comparison of queries is the comparison of non-empty results. If the results of two queries for a database instance do not indicate similarity or lack of similarity, we generate database instances so that each database query will have at least one database instance for which the query returns a non-empty result.

We developed two more methods of estimating the semantic similarity between SQL queries: Attribute Analysis and Vector Model. These two methods are efficient in terms of computation time and have no false negatives, so we use them in order to reduce the number of queries to be compared when searching for queries similar to the user query. In addition, we compare the accuracy of these two methods with Tree Edit Distance and Result Analysis, in order to show that the methods we developed are significantly more accurate than existing methods. Vector Model is actually an adaptation of a standard IR method for text comparison. In order to compare SQL queries and compare vectors in order to estimate their similarity, this method represents queries as vectors of terms. The traditional Vector Model is not suitable for comparing SQL queries and therefore we had to adapt it to the type of search we are conducting. Attribute Analysis was developed based on the idea that two queries are similar if they deal with similar sets of attributes. In order to compare two queries, we compare sets of attributes that appear in the conditions of the queries or attributes of the query result. This method of analyzing attributes is inefficient since the relevant attribute sets are usually small. This method is less efficient in terms of memory, but we present a recommendation system that helps users find similar queries to their queries, according to each of the two search semantics. At times, the user does not know the query to start with, and the recommendation system allows a search without having to write a query, according to the search details. In such cases, the search is based on extracts of natural language text or the results of queries over synthetic databases. All the natural language text is expressions and combinations of expressions that appear within conditions, expressions such as GROUP BY or ORDER BY and expressions that are formed from aggregation functions ORDER BY or GROUP BY, and we select the best expression among them.

We conducted the following experiment to estimate the usability of our methods, we ran the experiment separately for semantic and syntactic similarity. We used all the methods to classify all database queries according to their similarity to the given query. We found that Attribute Analysis and Vector Model have a higher accuracy than the other methods, and that they are suitable for our type of search. In order to compare SQL queries, we used the concepts of the comparison of text and comparison of vectors. The Vector Model is actually an adaptation of a standard IR method for text comparison. In order to compare SQL queries and compare vectors in order to estimate their similarity, this method represents queries as vectors of terms. The traditional Vector Model is not suitable for comparing SQL queries and therefore we had to adapt it to the type of search we are conducting. Attribute Analysis was developed based on the idea that two queries are similar if they deal with similar sets of attributes. In order to compare two queries, we compare sets of attributes that appear in the conditions of the queries or attributes of the query result. This method of analyzing attributes is inefficient since the relevant attribute sets are usually small. This method is less efficient in terms of memory, but we present a recommendation system that helps users find similar queries to their queries, according to each of the two search semantics. At times, the user does not know the query to start with, and the recommendation system allows a search without having to write a query, according to the search details. In such cases, the search is based on extracts of natural language text or the results of queries over synthetic databases. All the natural language text is expressions and combinations of expressions that appear within conditions, expressions such as GROUP BY or ORDER BY and expressions that are formed from aggregation functions ORDER BY or GROUP BY, and we select the best expression among them.
לכן ניסוח שאילתות עשוי להיות ממורכב, מסד הנתונים הינו בעל סכימה בארגונים גדולים, לעתים קרובות הנתונים, ובמקרים רבים מסד מסובך וקשה. לרוב, שאילתות נכתבות ונקבעות על ידי מקצועיים מכירים את נתונים מסד. שאילתות מורכבות שוב ושוב על ידי אפליקציות. אולם, ישנם מקרים שבהם משתמשים בסביבה כזו - אימות נתונים בזמן אמת מעל כמויות גדולות של נתונים. ובמקרים רבים, שאילתות גדולות מחברות המשתמשים יכולים לספק את שאילתות רלוונטיות, דומות לשאילתת שאותה הם מעוניינים לחשב, במסדי נתונים הקיימים של מסד הנתונים. המשמעות היא לסייע בלמידה ואנליזה של המערכת. יכולת למצוא שאילתות דומות לשאילתת נתונה, למשל, מנהל בחברה יכול לבחון האם שאילתות מסוג כלשהו הורצו במערכת lately, למשל על מנת לוודא שהעובדים אינם משתמשים בנתוני החברה באופן בלתי הולם.

היכולת למצוא שאילות דומות לשאילתת נתונה עשויה להיטמע סיטואציות שונות. למשל, משתמש מגלב באולינה מסוימת. להיות שימושית כאשר משתמש מגלב באולינה מסוימת, הוא יכול למצוא האם ומתי חושבו שאילות דומות לשאילתה שגויה, על מנת לנסות ולהקטין את הנזק הנגרם על ידי תשובות שאינן נכונות. שימוש אחרמציאת זוגות של שאילות דומות הוא לגילוי ומניעת העתקות של עבודות הבית בקורס מדעי נתונים.

הباحث עשה בפתקולשת למדעי המחשב בחנויות פרופ' ירון קנזה.

תודה

למנתה שליע, פרופ' ירון קנזה, זו היחידה זוהי דרך נוחה להמלות פרמי.

אני מודה לategori על התמיכת הבספית והサポート בשיתוף.
הכוגנה ייעלה לחיפוש שאילתות

חיבור על מחקר

לשם מילויה חלקי של הדורותחצ החובלת תואר מ↲יגטר לימי
묭י המנшение

הוגרני בורודין

הוגש לפנינו התכניות – מدرك טכנולוגי לישראל

ספטמבר 2014 חינה אללה תשע”ד
הכוונה יעילה לחיפוש שאילתות

גרגורי בורודין