Query Matching for Report Recommendation

Veronika Thost
TU Dresden
Faculty of Computer Science
01062 Dresden, Germany
thost@tcs.inf.tu-dresden.de

Konrad Voigt
SAP AG
Chemnitzer Strasse 48
01187 Dresden, Germany
konrad.voigt@sap.com

Daniel Schuster
TU Dresden
Faculty of Computer Science
01062 Dresden, Germany
daniel.schuster@tu-dresden.de

ABSTRACT

Today, reporting is an essential part of everyday business life. But the preparation of complex Business Intelligence data by formulating relevant queries and presenting them in meaningful visualizations, so-called *reports*, is a challenging task for non-expert database users. To support these users with report creation, we leverage existing queries and present a system for query recommendation in a reporting environment, which is based on query matching. Targeting at large-scale, real-world reporting scenarios, we propose a scalable, index-based query matching approach. Moreover, schema matching is applied for a more fine-grained, structural comparison of the queries. In addition to interactively providing content-based query recommendations of good quality, the system works independent of particular data sources or query languages.

We evaluate our system with an empirical data set and show that it achieves an F_1-Measure of 0.56 and outperforms the approaches applied by state-of-the-art reporting tools (e.g., keyword search) by up to 30%.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*information filtering, selection process*

Keywords

Query Matching; Query Recommendation; Business Intelligence

1. INTRODUCTION

Today, the collection and processing of large amounts of data has become important in various fields. In line with this is the need for extracting and preparing the often complex data for analysis and presentation; and, reporting has become an essential part of everyday business life, recently. For that reason, however, reporting is more and more done by business experts themselves (i.e., instead of database experts) [13]. For them, reports represent a means to visualize various facts and complex calculations on their specific data for presenting it to others in everyday business.

Reporting thus makes out an integral part of business. Especially for non-expert database users, however, the creation of reports represents a major challenge. In particular, the report query must cover the relevant Business Intelligence (BI) data and therefore include more complex constructs such as filters and groupings in SQL. These specifics of the data are then to be reflected in appropriate visualizations, which convey the knowledge as intended.

To ease the complicated task of report creation, the reuse of reports (i.e., applying the same visualization or query on different data) has become common practice. People look for and try to adapt existing queries or copy visualizations of colleagues. However, doing so, they encounter several problems. The first challenge are the large amounts of existing reports, where the finding of relevant samples to adapt is hard. It is also not clear how the comparison of reports and queries in different formats and languages—to find good samples—can be done independently of specific tools. In addition, semantic differences and inconsistencies between queries need to be considered. There are, for example, different design possibilities to describe semantically equivalent concepts. Moreover, the data a query describes may be domain-specific, which is reflected in the language used, too. It is not possible to cope with these syntactic and semantic differences without a fine-grained and comprehensive consideration of the queries.

Today's reporting tools provide only few means to alleviate the difficulties of the users; for instance, with visual query builders [2]. Additionally, they comprise add-ons such as keyword search [2] or auto completion mechanisms [3, 2], and recommender systems are investigated [9, 16]. Nevertheless, predefined example queries are used in practice [6, 4].

In this paper, we take up this practice of the users and propose a system for automatic recommendation of proven existing queries, which is integrated in the reporting environment Remix [22]. In this way, we encourage collaboration and sharing of reports, motivate reuse, and provide substantial support for the everyday reporting tasks in business. When a user is preparing a data source, Remix compares the query of the user to queries from existing reports and recommends similar queries and corresponding visualizations to the user. The key contribution of this paper is the automated comparison of queries, the so-called *query match-*

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.

http://dx.doi.org/10.1145/2505515.2505562.

ing. Instead of a simple syntactic comparison (e.g., keyword or text search) as it is applied by today's reporting environments, Remix provides a combined, *content-based* (i.e., focusing on the information contained within the queries) approach for matching queries. Working on an abstract representation of the queries, our system is independent of specific report formats that use particular query languages. To handle the large numbers of reports to be considered in real-world reporting environments, our system first uses a scalable filtering technique that applies a coarse-grained syntactic query comparison. The specific, nested structure of report queries is subsequently addressed with a fine-grained matching step based on schema matching.

We implemented this query matching approach in the Remix prototype and evaluated it on two empirical datasets to demonstrate that Remix interactively provides query recommendations of good quality (i.e., it correctly identifies queries to be recommendations and ranks them accordingly). In particular, it achieves an F_1-Measure of 0.56 and with up to 30% outperforms the approaches applied by state-of-the-art reporting tools.

The paper is structured as follows: Section 2 describes a motivating example of reporting in Remix, and Section 3 gives an overview of related work. We present the query matching part of the Remix system in Section 4 and subsequently describe the evaluation in Section 5. Finally, Section 6 concludes the paper.

2. MOTIVATING EXAMPLE

In this section, we describe a motivating example of using Remix to support query formulation. Consider the following SQL query, from now on called *original query*, which was created in a real reporting environment and looks for the revenue per industry (selected with column BRAN1) of company org.ai in Chicago. It might be issued by a new employee who wants to create a corresponding report.

```
SELECT SUM("REVENUE") AS "REVENUE",
"BRAN1.description" AS "BRAN1"
FROM "_SYS_BIC"."org.ai/CA_AI_RELATIONSHIP"
WHERE ( "LOC01" = 'CHICAGO')
GROUP BY "BRAN1.description"
ORDER BY "REVENUE" DESC
LIMIT 100
```

Note that the names used are rather cryptic and one has to be familiar with the database (which can contain several tables with hundreds of columns) or browse through it to formulate such a query. Also note that we restrict the examples in this paper for reasons of space and clarity—queries of real reports are usually much larger (e.g., a word count of 550 is not unusual).

Assume now, though being entirely valid, the above query does return an incomplete result obviously missing tuples. If similar queries were created in the company before (e.g., in other departments) and stored within the system, Remix—upon request—would provide them as recommendations. Two such recommendations are given below.

```
SELECT SUM("PROFITMARGIN") AS "PROFITMARGIN",
"BRSCH.description" AS "BRSCH"
FROM "_SYS_BIC"."org.ai/CA_AI_RELATIONSHIP"
WHERE ( "LOC01" = 'Phoenix' OR "LOC01" = 'PHOENIX'
OR "LOC01" = 'PHoenix')
GROUP BY "BRSCH.description"
ORDER BY SUM("PROFITMARGIN") DESC
LIMIT 500
```

```
SELECT SUM("GROSSREVENUE") AS "GROSSREVENUE",
SUM("REVENUE") AS "REVENUE",
"BRSCH.description" AS "BRSCH"
FROM "_SYS_BIC"."org.ai/CA_AI_RELATION_MONITOR"
WHERE ( "NAME1" = 'Lear Technologies Ltd.')
GROUP BY "BRSCH.description"
ORDER BY "GROSSREVENUE" DESC, "REVENUE" DESC
LIMIT 100
```

Although these recommendations consider different sections of the company's database, they are intuitively similar to the original query. For instance, the first query contains a hint of how the filter condition should be adapted in order to include all the relevant data. Obviously, the spelling of city names varies within the database of org.ai. The second recommendation shows that there is a column GROSSREVENUE in the database, which our employee decides to include in his report, too. The result of adapting the original query is the following.

```
SELECT SUM("GROSSREVENUE") AS "GROSSREVENUE",
SUM("REVENUE") AS "REVENUE",
"BRAN1.description" AS "BRAN1"
FROM "_SYS_BIC"."org.ai/CA_AI_RELATIONSHIP"
WHERE ( "LOC01" = 'CHICAGO' OR "LOC01" = 'Chicago'
OR "LOC01" = 'CHicago')
GROUP BY "BRAN1.description"
ORDER BY "GROSSREVENUE" DESC, "REVENUE" DESC
LIMIT 100
```

Finally, Remix would recommend a suitable visualization to our employee and integrate the different components into a complete report.

Note that the tasks for remix are complex. First, it has to cope with heterogeneous queries of arbitrary reports (e.g., in different formats) and to process the possibly large numbers of queries efficiently. Also, the matching should be fine-grained and concentrate on different kinds of information contained within the queries. For instance, a simple keyword search for "_SYS_BIC"."org.ai/CA_AI_RELATIONSHIP" would not retrieve the second recommendation, because the latter does not contain exactly this relation name. However, the recommendation is definitely similar to the original query and, if included, would improve the quality of the result.

In summary, the challenges for a query recommendation system for Remix are to process the (1) heterogeneous and (2) large numbers of report queries in the repository efficiently in order to (3) provide content-based query recommendations of good quality. Thus, our requirements extend those of existing systems, which are described in the next section.

3. RELATED WORK

In this section, we describe related work in the fields of content-based query recommendation, auto completion, and matching. We further review a matching technique that can be applied for matching queries.

3.1 Query Recommendation

The automated recommendation of similar queries is a very recent area of research. Database management systems (DBMS) generally do not offer such functionality. Nevertheless, they often maintain query repositories [4, 6] with well-

defined query examples or query logs [2] where the user can manually search for relevant queries—without being pointed to selected queries.

In [15], Khoussainova et al. describe a query log with enhanced browsing functionality, an extended keyword search; but the identification of relevant queries in the log, which is based on rather low-level matching techniques (i.e., the functionality provided by the underlying DBMS), is only referred to marginally.

The goal of the QueRIE recommender system [9] is to determine queries in the log that fit into the session of the current user. Thereby, the query similarity is computed by comparing the parts of the database addressed by the queries. However, though considering the queries on a more fine-grained level, by parsing and decomposing them, QueRIE basically performs a string comparison.

Similar methods are applied by content-based recommender systems for multidimensional databases, which consider properties of the query as a whole (e.g., result measures, the selection, etc.) and compute the similarity value for two queries by using string-based similarity measures [16]. Note that, in this context, Marcel et al. [16] raise an important issue by stating that the quality of the recommendations needs to be assessed based on real users and cases, which has not been done, to date.

As with query recommendation, query logs are considered as basis for recommendations by auto completion approaches, which are another kind of service provided on top of DBMS.

3.2 Auto Completion

In contrast to recommender systems, which recommend entire queries, auto completion systems focus on recommending query parts. Several commercial systems provide auto completion regarding tables or columns [3, 2], but more elaborate completion approaches are still subject of research.

Yang et al. [24] automatically complete a query with join operations based on the requested attributes. The paths of joins are found by mining the query log and applying a predefined quality measure (e.g., the share of queries in the log that contain the selected path).

SnipSuggest [14] is an interactive query completion system, which takes various parts of the query (e.g., tables and selection conditions) into consideration. By keeping the queries from the query log within a single graph structure capturing structural relations between the queries, valid and probable completions can be recommended to the user.

FIMIOQR [13] is an auto completion system for multidimensional queries. Similar to the approach of SnipSuggest, its recommendation of query parts for a partial query is based on mining the query log for the parts that most frequently complement those of the partial query.

3.3 Others

Query matching has been studied early in the area of query optimization [12]. However, these approaches are usually exhaustive—and thus complex—comparisons regarding only a single DBMS and rather simple queries (e.g., Select-Project-Join queries). Recent works (e.g., [21, 25]) also apply more efficient methods, but usually focus on physical properties (e.g., the cost of execution) instead of the intent of the queries.

Apart from the fields already mentioned, different kinds of queries (e.g., keywords or natural language sentences) are completed, compared, and translated in several areas such as keyword search [18] and natural language interfaces to databases [17] . However, such queries are of a nature very different from complex report queries.

The approaches presented above usually address the specific syntax and semantics of queries by parsing them, and often also normalizing them, before processing. Instead of considering the query parts individually, the auto completion approaches described explicitly take into account structural similarities between queries by using graphs for query representation. However, these graphs would be considerably greater in size and more complex if more heterogeneous queries of arbitrary complexity and data sources were considered. Also, the comparison of queries is simplified in the context of a single DBMS, because term equality can be used for a straightforward comparison of query parts that nevertheless leads to results of good quality. Hence, we need to consider further methods for Remix.

3.4 Schema Matching

To obtain a more general and at the same time structural comparison of the queries, we consider schema matching. Schema matching is a research area in the database community, which contains a large body of work on various matching approaches (e.g., based on linguistic methods, structural analysis, or domain knowledge). Further, there are several matching systems [8], which could be applied for matching queries, too. However, for query matching the existing systems need to be adapted: the missing full automation of the matchers, which is caused by a lack in preciseness and completeness, and their ignorance of the semantics of the query language would otherwise lead to unsatisfactory results. W.r.t. the scalability required for Remix, the usually quadratic (or larger) complexity of schema matching needs to be addressed, too.

Section 2 described the challenges for a query matching system to interactively provide content-based recommendations of good quality within Remix. However, this section showed that the existing approaches usually are either efficient (for interactivity) or precise (for quality). For that reason, we combine the methods applied by state-of-the-art approaches, extend, and augment them, for creating a query recommendation system for Remix. This is described in the next section.

4. QUERY MATCHING IN REMIX

In this section, we present the query recommendation part of Remix, which is based on a combined query matching approach. Remix itself is a reporting environment with special focus on report recommendation and integration, as described in Section 2. It maintains a report repository, where it stores reports created in the past and also separates the different components of the reports: links to data sources, queries regarding such data sources, and visualizations of these queries. Upon request, it matches a given query of a user to the queries in the repository and provides the most similar queries as recommendations to the user.

This query matching procedure is shown in Figure 1. Its input consists of the user-query, and the system itself contains the set of possible query recommendations. First, the queries are pre-processed to abstract from specific query lan-

Query	*REL_NAMES*	*PRO_ATTRS*	*SEL_ATTRS*
q_0	_SYS_BIC.org.ai/CA_AI_RELATIONSHIP	REVENUE,BRAN1.description	LOC01
r_1	_SYS_BIC.org.ai/CA_AI_RELATIONSHIP	PROFITMARGIN,BRSCH.description	LOC01,LOC01,LOC01
r_2	_SYS_BIC.org.ai/CA_AI_RELATION_MONITOR	GROSSREVENUE,REVENUE,BRSCH.d...	NAME1

Table 1: Selected features of the example queries

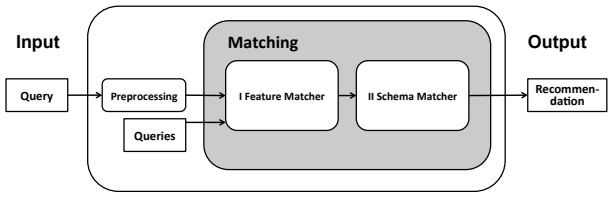

Figure 1: The query matching approach of Remix

guages. This pre-processing has the goal to prepare and facilitate the matching phase. In order to enable a comparison, it parses queries in different source formats and languages and translates them into a common representation based on relational algebra. Note that the current implementation of our system supports only SQL queries, though.

The main processing is then performed by a combination of two matching algorithms, also called *matchers*. In particular, we apply the *Feature Matcher* and the *Schema Matcher*. The former only applies a coarse-grained comparison of the queries and serves as scalable filter for the possibly large amount of queries. The latter, in contrast, performs a fine-grained and, in particular, structural comparison to capture subtle differences and similarities between the queries.

The output of the system is an ordered subset of the best recommendations. In the following, we give details about the two matchers.

4.1 The Feature Matcher

The *Feature Matcher* is applied in the beginning of matching in order to filter out queries that are not relevant for the final recommendation, because they are not similar to the original query. It is based on *similarity search*, a matching technique that, given a set of objects, looks for the most similar objects (i.e., queries, in our case). In this context, we introduce *features*, the characteristics of the queries that are compared by the matcher. Then, we give a detailed overview of the comparison and subsequent similarity calculation for the queries, the actual query matching.

4.1.1 Features

For a coarse-grained comparison of the queries, the *Feature Matcher* concentrates on general characteristics of queries, which can be extracted for every query (e.g., the data sources it addresses, the functions it applies, etc.). We extract these properties from the queries and record them textually as so-called *features*, as it is demonstrated in Example 1. A complete overview of the features considered in Remix is given in Appendix A.

Example 1. *Consider the first version of the query given in Section 2. Table 4 shows several features (i.e., the names of the relations addressed and those of the attributes in the projection and selection) we extract from it (q_0) and from the two recommendations (r_1 and r_2) proposed for q_0.*

Since we record the features of the queries textually, especially those of the queries in the repository, and because we use equality as comparison criterion, we can store them easily after extraction and apply index search for comparing them.

4.1.2 Comparison and Similarity Calculation

We apply similarity search as a textual comparison of features such that it can be realized with an inverted index [7], a data structure that generally supports fast full text search on large databases (e.g., a set of documents). With this application of index search, which achieves logarithmic complexity, the *Feature Matcher* efficiently realizes the comparison of the extracted features and can be used as scalable filter for queries not relevant for the final recommendation. For that, the features of the queries to be compared are extracted and stored separately, with references to the respective queries, in an inverted index. For the comparison of two queries, the index then is queried by transforming one of the two queries in a *search query* for the index. This is demonstrated in Example 2.

Example 2. *Below, q_0 has been transformed in a query for index search. The index considered uses the three features REL_NAMES, PRO_ATTRS, and SEL_ATTRS for indexing. This indexing of the repository queries is illustrated for r_1, which is shown how it is represented in the index.*

q_0:
REL_NAMES:_SYS_BIC.org.ai/CA_AI_RELATIONSHIP
OR *PRO_ATTRS*:REVENUE
OR *PRO_ATTRS*:BRAN1.description
OR *SEL_ATTRS*:LOC01

r_1:
REL_NAMES:_SYS_BIC.org.ai/CA_AI_RELATIONSHIP
PRO_ATTRS:PROFITMARGIN,BRSCH.description
SEL_ATTRS:LOC01,LOC01,LOC01

Since the index-based comparison is based on the text recorded in the features, the function determining the similarity between two queries has to concentrate on this information. We consider an extended version of term frequency-inverse document frequency (TF-IDF), the approach common in text search, and, as a basis, use the formula described in detail in [5], which is applied in Apache Lucene [1], a library for index search. Further, we adapt and extend the approach of Apache Lucene in three ways, which has been shown to be of advantage [23]:

- We consider a simplified version of the formula in [5], because we omit the boosting of specific terms or documents (i.e., queries). The boosting of specific information would make assumptions about the input (i.e.,

what parts are more important), which generally cannot be made w.r.t. an arbitrary set of queries.
- However, in order to stress similarities between the queries, the similarity calculation only considers features that occur at least once in the search query and one other indexed query. Considering a particular set of queries, we thus get more distinct similarity values.
- To relativize the directionality of the similarity calculation, we restrict the similarity values to be in the interval [0..1] and can hence adapt them to become symmetric. For that reason, we divide the value given by the formula in [5] by the similarity of the search query to itself and take the average of the two values for a pair of queries (i.e., the repository query to be compared is considered as search query, too, and the user-query is indexed). Note that this also makes the results of different matchers comparable.

Example 3 demonstrates the similarity calculation.

EXAMPLE 3. *Consider again the queries q_0, r_1, and r_2. In this example, a similarity value is computed for the queries r_1 and r_2 regarding q_0 (i.e., we assume, our index contains only these three queries). For the corresponding formula $fscore$, refer to [5]. First the normalizing factor, common for all values, is calculated as*

$$\begin{aligned} qNorm(q_0) &= \frac{1}{\sqrt{idf(\_SYS\ldots) + idf(REVENUE)^2 + \ldots}} \\ &= \frac{1}{\sqrt{3 * (1 + \ln \frac{3}{2+1})^2 + (1 + \ln \frac{3}{1+1})^2}} \\ &= 0.45 \end{aligned}$$

Then, the individual values are calculated as for the two examples below:

$$\begin{aligned} fscore(q_0, r_1) &= \frac{2}{4} * qNorm(q_0) \\ &\quad * (idf(\_SYS\ldots)^2 + \sqrt{3} * idf(LOC01)^2) \\ &= 0.61 \\ fscore(q_0, r_2) &= \frac{1}{4} * qNorm(q_0) * idf(REVENUE)^2 \\ &= 0.11 \end{aligned}$$

Similarly, such values can be computed for all queries stored within Remix (i.e., q_0, r_1, and r_2, in our case) leading to the matrix shown below. Note that the values calculated above correspond to the entries in the first row of the matrix.

$$\begin{array}{c} \\ q_0 \\ r_1 \\ r_2 \end{array} \begin{array}{c} \begin{array}{ccc} q_0 & r_1 & r_2 \end{array} \\ \begin{pmatrix} 2.23 & 0.61 & 0.11 \\ 0.44 & 2.56 & 0.11 \\ 0.17 & 0.17 & 1.99 \end{pmatrix} \end{array}$$

After adapting the matrix regarding the interval [0..1] (i.e., all values of a row are divided by the maximum of that row) and taking the average of all matrix entries that should be symmetric, the matrix looks as follows.

$$\begin{array}{c} \\ q_0 \\ r_1 \\ r_2 \end{array} \begin{array}{c} \begin{array}{ccc} q_0 & r_1 & r_2 \end{array} \\ \begin{pmatrix} 1.00 & 0.22 & 0.07 \\ 0.22 & 1.00 & 0.07 \\ 0.07 & 0.07 & 1.00 \end{pmatrix} \end{array}$$

Note that the normalization of the values described above would not be necessary since we use the *Feature Matcher* in Remix only as filter. Hence, the ranking (obtained by function $fscore$) would be sufficient for our purpose, which is to forward the best ranked queries as recommendations to the *Schema Matcher*. Also, the normalization requires the original query to be contained within the index. Thus, the index has to be recreated if this is not the case. Nevertheless, the normalization has been shown to be clearly beneficial for the quality of our system.

In particular, a query r is eliminated and not considered further as recommendation for a query q if the similarity value calculated by the *Feature Matcher* for a pair of queries (q,r) is 0. In addition, the *Feature Matcher* can be configured with one of two filter-parameters, maxN or threshold. Configured with a maxN n, it ranks the queries after calculating the preliminary similarity values as shown in Example 3 and forwards the best n queries to the *Schema Matcher*. Configured with a threshold t (e.g., obtained using training data), it forwards only those queries to the *Schema Matcher* for which the computed value is greater than or equals t. Note that for the latter, the normalization has to be carried out.

The *Feature Matcher* serves well to reduce the large set of possible recommendations in the system repository. However, the features only partially capture the syntactic information contained in the queries. Especially, structural relations regarding individual parts of the query (e.g., nested selection conditions or properties of subqueries, which count to those of the surrounding query) are not reflected and thus not considered during similarity calculation, because every feature is recorded only once and in a rather simple textual form. In order to take also more subtle differences and similarities between queries into account, we additionally apply the *Schema Matcher*, which explicitly regards the query parts in the context of the structure.

4.2 The Schema Matcher

The *Schema Matcher* is applied for a more fine-grained comparison and also determines the order of the final recommendations by calculating similarity values. It is based on different methods from schema-matching, which compares tree structures. Therefore, we consider the parse trees of the queries as the information to be matched. For comparing these often deeply nested structures of report queries, schema matching algorithms, also *schema matchers* are particularly suitable.

4.2.1 Query Schemas

The schema structure taken in this work, in essence, is the parse tree of the queries. It is based, however, on the abstract representation. The schema we create is further augmented by making additional information (e.g., the name of the query and data types) explicit. For instance, the schema for q_0 is displayed below.

```
Schema q0[ComplexRelation]
  -Projection[Projection]
    -REVENUE[Attribute]
    -BRAN1.description[Attribute]
  -Group[Group]
    -BRAN1.description[Attribute]
    -aggregationAttributes[List<ComputedAttribute>]
      -REVENUE[ComputedAttribute]
          ⋮
```

⋮
-REVENUE[Attribute]
-SUM[AggregationOperator]
-Sort[Sort]
-REVENUE[Attribute]
-Selection[Selection]
-condition[BinaryCondition]
-LOC01[Attribute]
-'CHICAGO'[CHAR]
-EQ[ConditionOperator]
-_SYS_BIC.org.ai/CA_AI_RELATIONSHIP[Relation]

Since the schemas we create for the queries contain different sorts of information, the *Schema Matcher* applies not only one but several distinct schema matchers for comparing the queries.

4.2.2 Comparison and Similarity Calculation

The application of schema matching especially targets the nested structure of report queries. Schema matching algorithms, so-called *schema matchers*, map the elements of two schemas that correspond semantically to each other. For that, the cartesian product of element-to-element similarity values v, with $v \in [0..1]$, is computed between the schemas. It constitutes a matrix of pair-wise similarity values. The values in this matrix determine the degree of correspondence between the pairs of elements. Elements that correspond to each other are called *matches* for each other.

For such a structural comparison of two queries, the *Schema Matcher* makes use of several different schema matchers by applying an entire schema matching system, the Auto Mapping Core (AMC) [19]. The latter executes various schema matchers in parallel and aggregates their results based on different strategies. In preliminary experiments [23], a specific combination of three matchers, which are described below, turned out to best suit our query matching approach. All three of them are based on the Name Matcher, which compares the names associated with the schema elements and *"matches schema elements with equal or similar names"* [20] by applying, for example, established string similarity algorithms [10] or considering synonyms.

Weighted Name Matcher The Weighted Name Matcher proceeds similar to the Name Matcher but breaks down the element names into tokens.

Children Matcher The Children Matcher compares two elements w.r.t. their direct successors, or *children*, in the schema. Thus, it assumes two elements to be similar if their children correspond to each other. Note that, to determine the correspondence of the children, their similarity has to be known (e.g., it may be computed in advance with another matcher).

Leaf Matcher The Leaf Matcher matches two elements by comparing all those of their successors that are leaves (i.e., attributes in the schema). Hence, it classifies two schema elements as similar if the attributes finally descending from them are similar.

Note that the matchers described above represent a substantial improvement compared to the *Feature Matcher*, which only considers term equality. They would, for example, detect a correspondence between the two relation names "_SYS_BIC"."org.ai/CA_AI_RELATION_MONITOR" and "_SYS_BIC"."org.ai/CA_AI_RELATIONSHIP" considered in the example of Section 2. Moreover, the Children and Leaf Matcher regard relations between different query parts. Also note that we take into account the issues to be considered when applying schema matching to queries described in Section 3. By applying a combination of different matchers, the deficiencies of individual matchers, which only focus on a single characteristic of the schemas, can be diminished, and even an automated application leads to convincing results [23]. Further, the pre-processing helps to overcome the missing awareness of query semantics. And since we apply a scalable filter in advance, the number of match-tasks for the complex *Schema Matcher* can be kept constant.

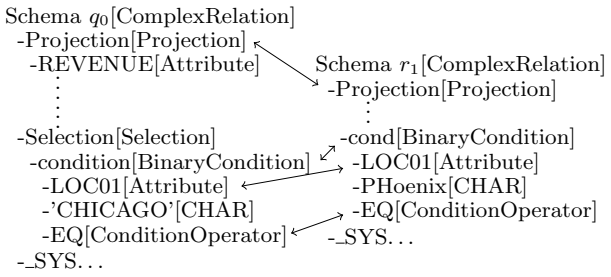

Figure 2: An extract of the mapping of the Weighted Name Matcher for queries q_0 and r_1

Figure 2 shows (an extract of) the result of applying the Weighted Name Matcher to compare q_0 and r_1. We compute such a *mapping* using the three matchers described above and then aggregate the mappings of the individual matchers by taking the average value for each pair of elements between the two query schemas (e.g., for mapping q_0 and r_1, we take the average for each pair where the first element is in the schema of q_0 and the second in the schema of r_1), as it is proposed in [11]. In order to match only relevant element pairs, we further consider only those pairs with a similarity greater than a so-called *selection threshold* [11] and set the remaining values to 0. This is demonstrated in Example 4.

EXAMPLE 4. *The below matrices show an extract of the mapping between q_0 and r_1 obtained by applying the Weighted Name Matcher before (top) and after filtering the values with a selection threshold of 0.7, which was shown to be of advantage in [23].*

	Proj.	⋯	*cond*	*LOC*01	*PHoenix*	*EQ*	*_SYS...*
Proj.	1.00	⋯	0.20	0.20	0.40	0.10	0.14
REVENUE	0.20	⋯	0.14	0.00	0.29	0.17	0.07
⋮	⋮	⋱	⋮	⋮	⋮	⋮	⋮
Sel.	0.60	⋯	0.22	0.11	0.22	0.11	0.20
condition	0.50	⋯	0.47	0.22	0.11	0.00	0.16
*LOC*01	0.20	⋯	0.00	1.00	0.14	0.00	0.13
'CHICAGO'	0.20	⋯	0.14	0.12	0.14	0.00	0.18
EQ	0.10	⋯	0.00	0.00	0.14	1.00	0.02
_SYS...	0.14	⋯	0.12	0.13	0.11	0.02	1.00

	Proj.	⋯	*cond*	*LOC*01	*PHoenix*	*EQ*	*_SYS...*
Proj.	1.00	⋯	0.00	0.00	0.00	0.00	0.00
REVENUE	0.00	⋯	0.00	0.00	0.00	0.00	0.00
⋮	⋮	⋱	⋮	⋮	⋮	⋮	⋮
Sel.	0.00	⋯	0.00	0.00	0.00	0.00	0.00
condition	0.00	⋯	0.00	0.00	0.00	0.00	0.00
*LOC*01	0.00	⋯	0.00	1.00	0.00	0.00	0.00
'CHICAGO'	0.00	⋯	0.00	0.00	0.00	0.00	0.00
EQ	0.00	⋯	0.00	0.00	0.00	1.00	0.00
_SYS...	0.00	⋯	0.00	0.00	0.00	0.00	1.00

Nevertheless, the resulting mapping of the query schemas still consists of several values instead of one value expressing the over-all similarity of the two queries. For that reason, the individual values are combined into one single value using a (slightly) adapted version[1] of the function for a combined schema similarity described in [11]. The resulting function basically takes the average of all similarity values greater than 0 and then takes the share corresponding to the elements that are matched w.r.t. all elements of the two schemas. Example 5 demonstrates its application.

EXAMPLE 5. *Consider again Example 4, for the result of the Weighted Name Matcher after the application of the selection threshold. Within that matrix only 4 entries are matches. In addition, there are 14 other matches, which are not displayed due to space restrictions. Apart from one match with a value of 0.75, all matches have a value of 1.00. Further, both the LOC01 and the EQ element in the schema of q_0 are matched three times, and the SUM element twice. All other elements appear in only one match. Thus, 13 elements of the source and 18 elements of the target schema, take part in the mapping. Let E_{q_0} and E_{r_1} denote the sets of elements in q_0 and r_1, respectively, with $|E_{q_0}|= 20$ and $|E_{r_1}|= 32$. Then, the corresponding similarity value for the queries is computed as:*

$$ssim(E_{q_0}, E_{r_1}, M_{q_0,r_1}) = \frac{17 * 1.00 + 1 * 0.75}{18} * \frac{13 + 18}{20 + 32} = 0.59$$

After limiting the set of possible recommendations with the *Feature Matcher* and applying the three schema matchers within the *Schema Matcher* the query recommendation system of Remix would suggest the recommendations ordered as follows.

$$S = \{(r_2, 0.72), (r_1, 0.59)\}$$

5. EVALUATION

In this section, we present an extensive, empirical evaluation of the query matching and recommendation in Remix. To this end, we implemented the query recommendation system prototypically in Java using amongst others the Apache Lucene Search Engine Library [1] and the schema matching system AMC [19] for the matching functionality. Moreover, we compared our approach, *Combined Matcher* in the following, to the methods applied by state-of-the-art tools by considering queries from real applications rated (w.r.t. their qualification as recommendations), amongst others, by real people.

5.1 Experiment Design

In particular, our experiments aim to show the following:

- The quality of both the *Feature Matcher* and the *Schema Matcher* described in Section 4 is better than methods provided by state-of-the-art systems (e.g., keyword search).
- The combination of the *Feature Matcher* and the *Schema Matcher* within the *Combined Matcher* outperforms the individual approaches. In particular, it increases the quality of the *Feature Matcher* and solves the scalability issues of schema matching.

The effectiveness of the individual approaches is shown by comparing the results they delivered to the results of two baseline approaches, the *String Matcher* and the *Text Matcher*. In particular the *String Matcher*, which computes the similarity value by relating the size of the largest common substring of two queries to the one of the longer of them, represents the keyword search applied in state-of-the-art DBMS. The *Text Matcher* maintains an index and performs general text search thereby calculating the similarity values based on TF-IDF.

To measure the quality of matching, the recommendations R determined by a system are to be compared to those classified as recommendations in a reference set R'. According to Do [11], there are three subsets describing the recommendations that were identified correctly, I, those that were falsely classified as recommendations, F, and the recommendations that were missed, M. Based on this classification, we consider the following three quality measurements, all ranging between 0 and 1, with 1 representing the best value:

Precision describes the accuracy of the result and is defined as

$$pr = \frac{|I|}{|R|} = \frac{|I|}{|I|+|F|}$$

Recall describes the completeness of the result (i.e., the share of correct recommendations that were found), and is specified as

$$re = \frac{|I|}{|R'|} = \frac{|I|}{|I|+|M|}$$

F_1-Measure is the harmonic mean of precision and recall (to neglect none of them in favor of the other) and specified as

$$F_1 = \frac{2 * pr * re}{pr + re}$$

For finding the best recommendations, however, it suffices to regard the (sub)list of actual recommendations of a system. Hence, we consider also the k topmost results using the so-called precision@k [7].

Precision@k describes the accuracy of the result regarding a given cutoff rank k=$|S|$, and is defined as

$$pr@k = \frac{|I \cap S|}{|S|}$$

The evaluation results presented are usually the average values for all recommendation tasks in both datasets (i.e., naturally, the number of recommendation tasks corresponds to the number of queries in a dataset). In addition, we considered scalability. Therefore, we measured the runtime during the evaluation executing all matchers on a usual office PC using an Intel Core 2 Duo, four gigabyte RAM and a Java 6 (32-bit) environment.

[1]Specifically, we consider the matches as being undirected (i.e., the computed matrix of similarity values is supposed to be symmetric) and instantiate the function with a selection function that takes the average of the similarity values of all match candidates.

Property	SQLT	SDSS
Number of queries	43	150
Recommendations per query >3/>5/>10/>15 (in %)	61/49/28/2	96/87/52/29
Min/Max/Avg query length (word count)	4/42/18	6/82/38
Share of nested queries	7/43	0/150

Table 2: Statistics of the test data

5.2 Datasets

We evaluate Remix over two datasets with rather different characteristics, shown in Table 2. To measure the quality of query matching, our empirical datasets do not only have to contain an appropriate number of real-world queries, but must also contain details about the similarity between these queries. According to Marcel et al. [16], such datasets did not exist at the time of our experiments. Hence, we conducted an empirical study [23] where database users rated the similarity of query pairs on a scale of 1 to 4.

To get an overview of the characteristics of the individual matchers, we first considered queries extracted from an SQL tutorial as dataset SQLT, which we tagged by ourselves. These queries contain rather different features of SQL and have very different intends. Most of these queries are about the business domain.

For the SDSS dataset, we chose a random sample of queries extracted from the query log of the Sloan Digital Sky Survey (SDSS) [6]. To obtain the corresponding similarity values, we conducted an empirical study. The SDSS maintains a large open database containing scientific data about the universe. Its query log contains millions of SQL queries issued by different users and also by bots. As preparation, we removed query duplicates, queries too long to be grasped by the survey participants in an acceptable amount of time (i.e., queries consisting of more than 350 characters), and erroneous queries, and took 150 queries as basis for the SDSS dataset. We further determined query pairs that neither query the same table or a corresponding view nor use common attributes as not to be similar. This reduced the original 22,500 similarity values to be found to about 9,000. In total, 121 people participated in the survey. About 50% of them reported to have basic SQL knowledge, 40% had deeper, and 10% only rudimentary knowledge of SQL. On average, the participants rated 82 queries. Conversely, the retrieved similarity values were averagely rated by 1.2 people, who in about 70% of the 1,307 cases with multiple votes agreed in their rating. Altogether, the study lead to a dataset of 150 queries with similarity values between all of them.

Although the datasets described above consider distinct domains and contain queries with rather different characteristics, they have to be regarded critically; certainly, they cannot reflect the diversity of the real world. And also the similarity ratings, though retrieved empirically, are subjective by nature. Nevertheless, the data is suitable to show the quality and performance of our combined query matching approach compared to the methods applied by state-of-the-art tools. The results of this comparison are given in the next section.

5.3 Results

In this section, we first present the evaluation results regarding the quality of the recommendations and then give an overview of the results concerning the scalability of the individual approaches.

5.3.1 Quality

First, we configured our system using one third of the sample data. This preliminary evaluation turned out that it achieves best results with a filter-parameter of $n = 20$. Recall, this means that the *Feature Matcher* ranks the queries after calculating preliminary similarity values and forwards the best 20 queries to the *Schema Matcher*. The latter recalculates the similarity values for the 20 queries, and the similarity values of the remaining queries, which are not considered by the *Schema Matcher*, are set to 0.

It turned out that all approaches considered have a very similar performance w.r.t. the top k recommendations. An overview of pr@5, pr@10, and pr@15 is given in Figure 3. Though the similarity of the results, only the *Schema Matcher* and the *Combined Matcher* achieve a pr@k greater than 0.6. The *Feature Matcher* shows a better quality than the baseline approaches, but the differences seem to be marginal; all values range between 0.5 and 0.6. This induces that with all matching approaches, about 50% of the top five recommendations are correct recommendations.

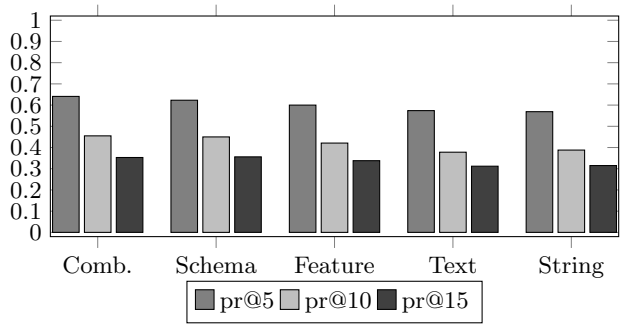

Figure 3: The evaluation of the query matchers w.r.t. recommendation quality

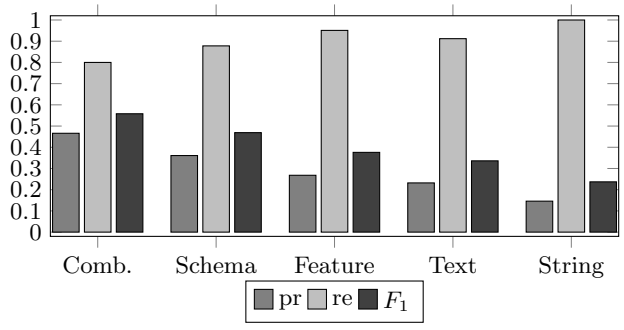

Figure 4: The evaluation of the query matchers w.r.t. matching quality

However, Figure 4 shows that precision, recall and F_1-Measure clearly show differences in the matching performance of the different matchers. Although all of the latter find the correct recommendations, which is indicated by the high recall, their over-all matching performance differs strongly. With F_1-Measure values of 0.38, 0.47, and 0.56 for the *Feature*, *Schema*, and *Combined Matcher*, respectively, the increase in the value makes up about 10%. With a precision of 0.47, only the *Combined Matcher* is near 0.5. This indicates that it is the only matcher where at least half of all recommendations determined are useful.

5.3.2 Scalability

Figure 5 gives an overview of the time the systems need for processing in dependence of the number of queries in the systems. Note that the data confirms the specifications of the complexity of the approaches, stated previously.

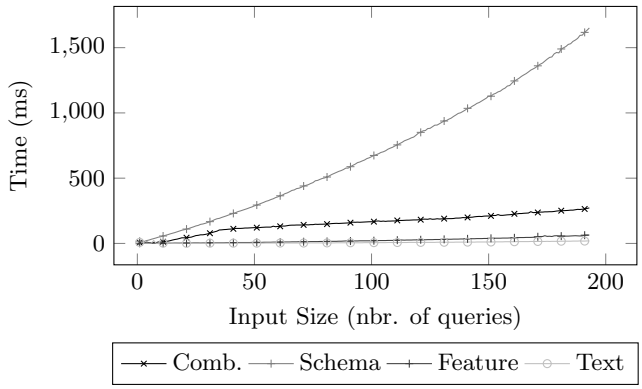

Figure 5: The evaluation of the query matchers w.r.t. scalability

The curves of the two baseline approaches both develop rather steadily with only a slight increase with growing input. Although it has to maintain the terms, the *Text Matcher* requires less time than the *String Matcher* (not shown here) —by using an index, it achieves logarithmic complexity. Similarly based on an index, the *Feature Matcher* shows the same development. However, since it is based on the abstract representation of the queries, the pre-processing required by the approach leads to an increase of processing time. Also the quadratic complexity of the schema matchers is reflected by the time they need for processing, which even increases along the quadratic parable. Finally, the *Combined Matcher* shows a logarithmic curve, too. Since the filter-parameter n sets a bound for schema-matching, its complexity is based on the indexing of the filtering.

6. CONCLUSION

In this paper, we presented a query recommendation system for the reporting environment Remix, which is based on query matching. Our system is motivated by the increasing complexity of reporting in Business Intelligence and the fact that this is often done by non-expert database users, today. To interactively provide content-based query recommendations of good quality to them, we proposed a hybrid approach of query matching combining an efficient and, in particular, scalable feature-based matcher as preliminary filter for the possibly large set of queries and a fine-grained structural matcher based on schema matching. Moreover, we evaluated our system on empirical datasets containing real queries and similarity ratings obtained from real users. Thus, we showed that it outperforms the methods applied by state-of-the-art applications by up to 30%.

Future work on the system includes to focus on the particularities of report queries (e.g., to consider also multidimensional queries and additional information about the reports the queries come from) and to refine the existing matchers. In particular, the comparison of the *Feature Matcher* currently based on term equality should be less restrictive (e.g., through clustering features with similar values). Further, it would be interesting to consider query semantics also for matching (e.g., to refine the ranking of syntactically very similar queries).

7. REFERENCES

[1] Apache lucene. http://lucene.apache.org/.

[2] Aqua data studio. http://www.aquafold.com/aquadatastudio.html.

[3] Databasespy - multi-database tool and sql editor. http://www.altova.com/databasespy.html.

[4] Gene ontology. http://www.geneontology.org/.

[5] Lucene 3.5.0 api. - class similarity. http://lucene.apache.org/core/3_6_0/api/all/org/apache/lucene/search/similarity.html.

[6] Sloan digital sky survey. http://www.sdss.org/dr8/.

[7] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[8] P. A. Bernstein, J. Madhavan, and E. Rahm. Generic schema matching, ten years later. *PVLDB*, 4(11):695–701, 2011.

[9] G. Chatzopoulou, M. Eirinaki, S. Koshy, S. Mittal, N. Polyzotis, and J. S. V. Varman. The querie system for personalized query recommendations. *IEEE Data Eng. Bull.*, 34(2):55–60, 2011.

[10] W. W. Cohen, P. D. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *IIWeb*, pages 73–78, 2003.

[11] H.-H. Do. *Schema Matching and Mapping-based Data Integration*. PhD thesis, University of Leipzig, 8 2005.

[12] A. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10:270–294, December 2001.

[13] R. Khemiri and F. Bentayeb. Interactive query recommendation assistant. In *DEXA Workshops*, pages 93–97, 2012.

[14] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. Snipsuggest: Context-aware autocompletion for sql. *PVLDB*, 4(1):22–33, 2010.

[15] N. Khoussainova, Y. Kwon, W.-T. Liao, M. Balazinska, W. Gatterbauer, and D. Suciu. Session-based browsing for more effective query reuse. In *SSDBM*, pages 583–585, 2011.

[16] P. Marcel and E. Negre. A survey of query recommendation techniques for data warehouse exploration. In *EDA*, pages 119–134, 2011.

[17] N. Nihalani, S. Silakari, and M. Motwani. Natural language interface for database-a brief review. *International Journal of Computer Science Issues*, 8:600–608, 2011.

[18] J. Park and S. goo Lee. Keyword search in relational databases. *Knowl. Inf. Syst.*, 26(2):175–193, 2011.

[19] E. Peukert, J. Eberius, and E. Rahm. Amc - a framework for modelling and comparing matching systems as matching processes. In *ICDE*, pages 1304–1307, 2011.

[20] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[21] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD Conference*, pages 249–260, 2000.

[22] M. Seguran, A. Senart, and D. Trastour. *remix*: A semantic mashup application. In *OTM Workshops*, pages 312–315, 2012.

[23] V. Thost. Calculating similarity of arbitrary reports. Master's thesis, TU Dresden, 2012.

[24] X. Yang, C. M. Procopiuc, and D. Srivastava. Recommending join queries via query log analysis. In *ICDE*, pages 964–975, 2009.

[25] J. Zhou, P.-Å. Larson, J. C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD Conference*, pages 533–544, 2007.

APPENDIX

A. FEATURES SUPPORTED IN REMIX

In Table 3, this section gives an overview of all features considered in Remix during query matching with the *Feature Matcher*. With the large selection of features we aim to capture as many characteristics as possible. For matching, however, we only consider the features appearing more often within the current set of queries, as it is described in Section 4.

Feature	**Description**
REL_NAMES	The names of the relations that are queried
PRO_ATTRS	The attributes in the projection
SEL_ATTRS	The attributes that occur in scope of a selection
SEL_COMP_OPS	Comparison operations that occur in scope of a selection
SEL_SQL_OPS	SQL-specific condition operations (e.g., 'BETWEEN') that occur in scope of a selection
SEL_PRED_OPS	Predicates (i.e., 'and' or 'or') that occur in scope of a selection
SEL_ARITHM_OPS	Arithmetic operations that occur in scope of a selection
SEL_AGG_OPS	Aggregation operations that occur in scope of a selection (e.g., in an SQL 'HAVING' clause)
REN_REL_NAMES	The names of the relations that are renamed
REN_ATTRS	The attributes that are renamed
EXT_NEW_ATTRS	The names of computed attributes
EXT_COMP_OPS	Comparison operations used for computing new attributes, the so-called *extension*
EXT_SQL_OPS	SQL-specific condition operations that occur in scope of an extension
EXT_PRED_OPS	Predicates that occur in scope of an extension
EXT_ARITHM_OPS	Arithmetic operations that occur in scope of an extension
SORT_ATTRS	The attributes that are used for sorting the result
GRO_GRO_ATTRS	The attributes for which a grouping is performed
GRO_ATTRS	The attributes that occur in scope of a grouping
GRO_NEW_ATTRS	The names of the new attributes resulting from a grouping
GRO_ARITHM_OPS	Arithmetic operations that occur in scope of a grouping
GRO_AGG_OPS	Aggregation operations that occur in scope of a grouping
JOIN_ATTRS	The attributes that are used in a join condition
JOIN_OPS	The join operations in the query
JOIN_COMP_OPS	Comparison operations that occur in scope of a join operation
JOIN_SQL_OPS	SQL-specific condition operations that occur in scope of a join operation
JOIN_PRED_OPS	Predicates that occur in scope of a join operation
JOIN_ARITHM_OPS	Arithmetic operations that occur in scope of a join operation
SET_OPS	The set operations that occur in the query
REN_COUNT	Count of rename operations
JOIN_COUNT	Count of join operations
SET_COUNT	Count of set operations
FUN	The functions that are used in the query
FUN_ATTRS	The attributes that are used as function parameters
CASE_COUNT	Count of condition statements (e.g., SQL 'CASE-WHEN' or 'IF-THEN-ELSE' statements)
SUB_COUNT	Count of subqueries
VALUES	The values that occur in the query

Table 3: The query-features considered in Remix