Exploring Databases via Reverse Engineering Ranking Queries with PALEO*

Kiril Panev, Sebastian Michel, Evica Milchevski, Koninika Pal
TU Kaiserslautern
Kaiserslautern, Germany
{panev|smichel|milchevski|pal}@cs.uni-kl.de

ABSTRACT

A novel approach to explore databases using ranked lists is demonstrated. Working with ranked lists, capturing the relative performance of entities, is a very intuitive and widely applicable concept. Users can post lists of entities for which explanatory SQL queries and full result lists are returned. By refining the input, the results, or the queries, user can interactively explore the database content. The demonstrated system is centered around our PALEO framework for reverse engineering OLAP-style database queries and novel work on mining interesting categorical attributes.

1. INTRODUCTION

The concept of rankings is ubiquitous; it exists in nearly all domains. Essentially, rankings allow focusing on a small subset of an exhaustively full list—usually a few top or bottom entries are of interest. Such small subsets represent the essence of the available data, worthwhile to look into. Instead of browsing through databases via OLAP cubes over predetermined dimensions to gain insights, we propose the usage of rankings to explore database contents. We put forward PALEO [6], our approach to reverse engineer OLAP-style database queries. Given an input result list, PALEO is able to efficiently determine input-generating SQL queries and can additionally be relaxed in order to find queries that generate rankings similar to the input within a certain distance bound. How is this useful for exploring data?

Consider a user Alice who needs to make up her mind which smartphone to buy next. Alice is favoring model X, model Y, and model Z, in this order. She is interested in finding explanatory queries and in fact populated rankings that resemble this ranking. PALEO tries to determine such queries, either explicitly reflecting Alice's preference or delivering queries and resulting rankings that are close to her ranking. Given the structure of the queries (perhaps translated to natural language) Alice learns about the categorical constraints and ranking criteria used. Given the computed rankings, Alice can further learn about other smartphones that perform perhaps even better, depending on how much PALEO is allowed to deviate from the original input ranking. Assume PALEO returned a ranking of {X, W, Y, Z} with constraints 'storage=16GB' and 'brand=Samsung', ranked by 'battery lifetime'. What can she learn from that and how can she proceed? She can remove the constraint on the make to get additional offers, she also learned that the model W appears feasible, too. Further, she changes the ranking criteria as 'battery lifetime' is not the most decisive criterion for her anyways, can distort the ranking slightly to see how generating queries are going to differ, etc.

Developing a system that allows working with rankings in such an exploratory fashion brings up several challenges that need to be addressed. First, subsecond response times are required to allow interactive data exploration. PALEO achieves this by precomputed statistics, decision trees, in-memory processing over a sufficient subset of the data, and low false positive rate in the candidate-query evaluation. Second, the system has to allow approximate answers to the user input, as it is not reasonable to assume that the identical ranking can be retrieved from the database content. Yet, the virtually exploding search space when allowing too much freedom needs to be kept tractable. We address this, by deeply incorporating distance-measure–based pruning into the candidate query generation. Third, for a specific input ranking there might be several explanatory SQL statements that yield the input when being executed. But not all syntactically close rankings and corresponding queries are equally interesting. Thus, a way to bring such candidates in an order that reflects a user-perceived notion of interestingness is required. To achieve this, PALEO employs novel insights from mining Web tables corpora to derive a classifier that is able to tell whether or not a non-numerical attribute is semantically meaningful to act as a constraint to the `WHERE` clause of a query.

In the following, we highlight the key components of PALEO, followed by a detailed discussion on the setup of the planned demonstration. Thereafter, we briefly discuss related work before we conclude this demonstration proposal.

2. SYSTEM OVERVIEW

PALEO is a system designed for exploring databases by reverse engineering OLAP-style top-k queries. As user input, the system consumes a top-k list L containing either one column $L.e$ of ranked entities or two columns that are capturing entities *and* their corresponding scores. Then, given a

*This work has been supported by the German Research Foundation (DFG) under grant MI 1794/1-1.

Proceedings of the VLDB Endowment, Vol. 9, No. 13

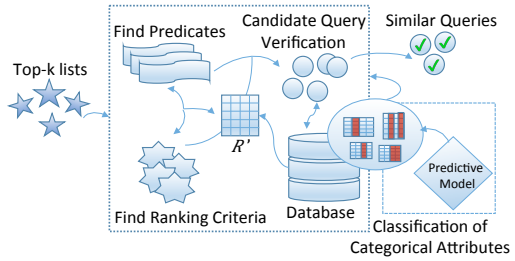

Figure 1: PALEO framework

database D with multiple relations R_i, where each relation R_i contains data from a single domain, PALEO efficiently and effectively determines queries Q_i that, when executed over the database, compute result lists that are **similar** to L. The found queries and corresponding top-k lists are ordered according to their similarity to the input ranking and also with respect to a human notion of interestingness; the latter will be discussed in more detail below. The similarity of result rankings to the input is controlled via a user-defined similarity threshold θ.

To quantify the similarity between the user-provided input list and the resulting top-k lists, we use the Footrule distance that is one of the predominant measures to compute similarity between ranked lists. Specifically, since top-k rankings are naturally only capturing a subset of all entities, we use the adaptation $F^{(k+1)}$ as proposed by Fagin et al. [3].

The system consists of the following steps, depicted in Figure 1:

- finding the predicate P in the `WHERE` clause of the reversed engineered queries Q.
- finding the ranking criterion according to which the ranked list (or a similar one) could be sorted.
- validation and ranking of the resulting queries and the corresponding resulting lists.
- classification of the database tables and their columns into interesting and non-interesting ones.

As the basis of all computation, we retrieve from the relation R all tuples where the entity is one of the entities in the input list L into one single relation. We refer to this table as R'. Furthermore, an inverted index is used as an auxiliary structure in identifying the domain of the entities in the provided top-k list, i.e., the base table R that contains the appropriate entities. Our system currently supports the following ranking criteria: $avg(A)$, $max(A)$, $sum(A)$, $sum(A+B)$, and $sum(A*B)$.

Predicates. As the user submits an input list to the system, and a similarity threshold θ, the first step of PALEO is to identify a set of *candidate predicates* by using the tuples in R'. We focus on predicates P of the form $P_1 \wedge P_2 \cdots \wedge P_m$, where P_i is an atomic equality predicate of the form $A_i = v$ (e.g., (team='Chicago Bulls')). If a query that precisely generates the input is to be determined, there must exist one tuple t_i for each entity with $P(t_i)$ evaluating to true. However, in case we want to find all queries corresponding to a similar list to L, in fact, all distinct attribute:value pairs in R' are atomic candidate predicates of size $|P| = 1$.

A naïve approach would simply take all candidate predicates and advance to the next step of finding ranking criteria. However, this would result in a drastically reduced performance, caused by a radical expansion of the set of candidate predicates.

PALEO reduces the search space by applying two techniques. First, it processes only the columns in R' classified as interesting with our pre-processing step. Only if no result is returned using these column, PALEO resorts to processing the non-interesting columns as well.

Second, by using the fact that the user expects as result only those queries resulting in lists **similar** to the input, i.e., those queries with Footrule distance $d(Q_L, L) \leq \theta$, we devised formulas that allow us to considerably reduce the number of candidate predicates, for reasonably small values of θ. First, we compute how many entities a list l_i must have in common with l so that $F(l_i, L) \leq \theta : \mu = \lceil 0.5 \times (1 + 2k - \sqrt{1 + 4\theta}) \rceil$. We also define how many of the top τ entities in L must be also ranked in l_i, so that $F(l_i, L) \leq \theta$: $\tau = k - min\{\frac{\theta}{2}, k\}$. We can in fact guarantee that any predicate in R' that is not satisfied by at least μ entities in L and all of the first τ entities, would result in a list that is within distance larger than θ. For details on this the reader can refer to [7].

Ranking Criteria. Using the set of identified candidate predicates, in the subsequent step, PALEO identifies suitable ranking criteria according to which the entities in the top-k lists are ordered. Identifying the candidate ranking criteria is efficiently done by leveraging the table R' which is held in-memory. PALEO combines all of the candidate predicates from Step 1 and the ranking criteria supported by the system into queries, and executes them on R' to find a candidate query matching the input list. If the scores of the ranked entities are given by the input, statistical methods and decision trees are used to efficiently identify the most promising column for a ranking criteria, in order to avoid executing all queries over R'. In the more general case when rankings without numerical scores are given as input, all combinations of numerical attributes and aggregate functions could potentially qualify as candidates. The candidate quires are further validated over R.

Validation of Candidate Queries. The resulting top-k lists that compared with the input list L are within a Footrule distance of θ are created as candidate queries Q_c and need to be further executed on the base table R. This is because there are in general entities outside R' that will qualify for the ranking and might distort it. The order of execution is done by ranking the candidate queries first in ascending order of $d(Q_L, L)$, i.e., the Footrule distance between the resulting list Q_L and the input list L and second, in descending order of the number of categorical attributes A_{cat} in the predicate of Q_c. Thus, for candidate queries with equal $d(Q_L, L)$, the ones with more categorical attributes in their predicate will be executed first, since they will be providing more interesting information to the users in their database exploration. The validation is done iteratively and in each step information is gathered to potentially eliminate forthcoming queries without executing them.

Classification of Columns. As mentioned earlier, not all queries reveal the same amount of information based on their structure, although they all might resemble the input ranking. To accommodate for this, we have developed a

classification model that is able to predict whether or not a column used as a constraint in the WHERE clause of a query is interesting with respect to human perception. PALEO then assigns scores to table columns according to their interestingness predicted by the classification model; higher scores are assigned to columns that are very likely to be part of a predicate in an interesting top-k query. Note that this preprocessing step is independent from the input list and the similarity threshold θ, it operates solely on statistical measures like information entropy computed over the values inside a database column.

For learning this classification model, a supervised learning approach, ν-SVM, is used on a training dataset that is built based on Wikipedia tables. We consider *a categorical attribute for an entity as interesting iff we find at least one Wikitable that is created by imposing a constraint over that categorical attribute*. Statistical measures like Entropy, Unlikeability, Peculiarity, and Maximum Coverage are employed to characterize the uncertainty in information, diversity and competitiveness of the training data. Three new statistical measures are also used as features that capture the diversity of data with respect to information content, uniform distribution, and variability. The feature vectors of the training dataset are extracted and used to learn the classification model. The accuracy of the classification model is described in [5]. Our combined scoring model reflects the tradeoff between result similarity in terms of the Footrule distance to the input ranking, and query-centric objectives like interestingness of column constraints etc.

3. DESCRIPTION OF DEMONSTRATION

To demonstrate our system and all its capabilities to the users we will use three datasets. The first dataset contains basketball statistics from the Database Basketball portal [1]. It contains per-season statistics of players and their teams, capturing the NBA and the ABA leagues from 1946 to 2009. In total, there are 3924 players and 95 teams. The database reports on various numerical attributes such as turnovers, rebounds, total points per game, assists, field goals, etc. Furthermore, we will also use a subset of the IMDB dataset [4], which contains info on movies, actors, genres, movie length, ratings, etc. The third dataset is the DBLP [2] database containing information about authors and publications focused in the field of computer science.

We have specifically chosen datasets that allow creating a large variety of top-k rankings, with different categorical constraints and ranking functions. This demonstration should further be not only informative but also fun for the users, specifically we expect the DBLP scenario to draw a lot of attention from the VLDB audience as most visitors will find themselves in the DBLP dataset.

Moreover, the domains of the datasets are diverse enough to make the demonstration interesting for people with different interests. We plan to engage the users using the following demonstration scenarios.

Head-to-head Comparison of Entities. In this demonstration scenario, users are asked to perform a comparison of specific entities of their choice. PALEO shows users how entities compare with each other based on different scoring functions and categories. In a head-to-head entity comparison, the performance of entities is agnostic to all other entities in the domain except the ones provided as input. Thus, the users will also be able to compare entities that normally do not belong in the same league, i.e., one is ranked significantly higher than another and therefore are not together in any top-k scenario. PALEO still provides an option of exploring these entities and shows how they compare against each other by ignoring the other entities that could (potentially) appear between them in the ranking. For instance, consider a scenario where the user wants to see how Mark Price compares with the best guards in NBA, even though this player is never really compared to the top guards that ever played in the NBA. This is because he cannot be compared to the elite, when considering the points scores, the assists made, or the rebounds, etc. However, the first query in Figure 2 will still be identified as a valid query, even though if considering all entities in the dataset, Mark Price is not in the top 100 players by points per game in a single season with 19.6 PPG. Furthermore, other interesting rankings can be found where Price is ranked even above Michael Jordan, e.g., for the free throw percentage or three point field goal percentage. In this way, the users can see the virtues of these non-elite players, i.e., the performance aspects where they can compete with the elite. In this scenario, PALEO only needs to utilize the table R' since it already contains all tuples for the entities in the input list. In this way, valid queries are identified very efficiently. The user can also ask to see the entities that are ranked between or above the input entities for a specific query. For instance, she wants to see how far away is Mark Price from the top 100 player ranked by points per game. Our system will do that, by additionally executing the query over the database. Then, the entire top-k list will be shown to the user.

Exploring Similar Lists. In this demonstration scenario we will show, given a ranked list of entities and a similarity threshold θ, a set of queries together with their top-k result lists and corresponding statistics. Figure 2 shows a screenshot demonstrating this scenario. The user can specify the top-k list by entering entities through an input field and then possibly rearranging them. We will prepare several interesting starting examples to assist users in getting started. Users can enter either only the entities or the entities and their scores. PALEO provides assistance when entering both fields; alternatively SQL queries can be used to retrieve a ranking that can be altered by users to get reverse-engineered. Users need to enter a similarity threshold, i.e., the maximum Footrule distance the input list can have to the lists produced by the identified queries. The system starts by pressing the "Find Queries" button. As output the system shows the user the Footrule distance of each query:list pair, the query execution time, and marks the interesting attributes in its predicate according to the classification of categorical attributes. The pairs with lower Footrule distance will be displayed more prominently by default, with the number of interesting categories as a second criteria. However, users can additionally change the ordering of the results: by the number of interesting categories in a query's predicate, or the query execution time. By inspecting interesting categories, users can see which categories were used in the WHERE clause of the query. For instance, in the example in Figure 2, the upper query uses only a constraint on the field position while the query below puts constraints to position as well as the league. Additional statistics of the entire process can be shown by pressing the "Under the

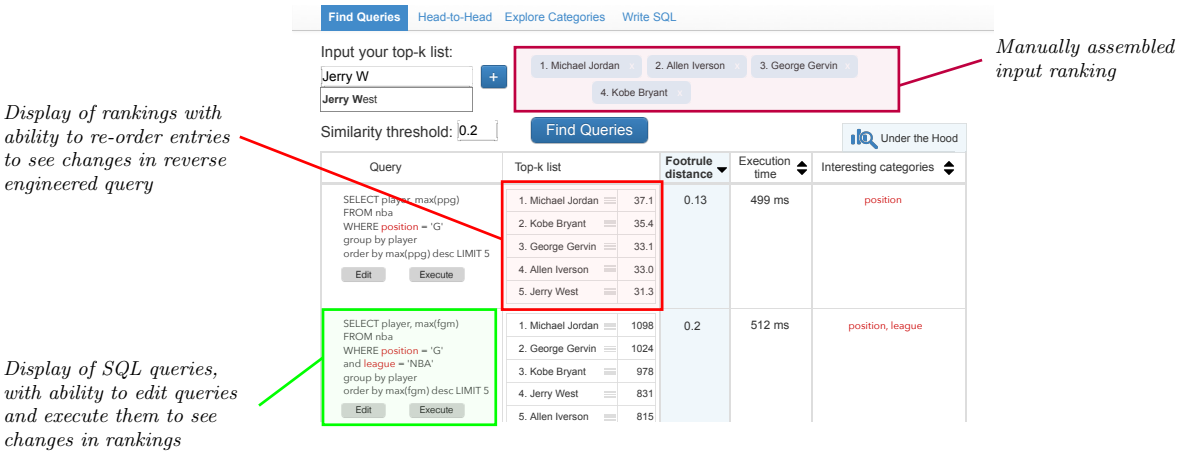

Figure 2: Screenshot of scenario for finding similar lists

Hood" button. The user can see the number of query executions needed until the first valid query is identified, the runtime of each of PALEO steps, the number of candidate predicates that were created and pruned, and the number of created candidate queries.

Classification of Database Columns. Understanding and exploring information in an arbitrary domain of knowledge is challenging as the complete domain usually contains a large number of entities (e.g., the NBA database contains 3924 players). A simple solution to comprehend such large data is to organize entities by ordering them using specific categories. The classification model integrated in PALEO provides guidelines to users in order to find interesting categories for a refined view on a subset of the entities in the example domains. Our model is able to tell whether or not categories are good categories to organize entities in. For example, our model suggests that, in the NBA dataset, a refined view on the subset of players based on the position they play(ed) is more interesting then a refined subset of players based on the university they went to or their birth date. Users can also have a quick overview of all tables in the database where the table columns are highlighted when the predictive model classifies them as interesting. By clicking on specific values of interesting categories, the users can dive into the data by adding the corresponding constraint. For more insights on table columns, PALEO displays statistical characteristics on how the categorical values are distributed over the entities, how their distribution differs from the uniform distribution, the dominating categorical values, etc.

4. RELATED WORK

Reverse engineering queries has been gaining popularity as a research topic recently. Given a database D and a query output $Q(D)$, Tran et al. [9] try to find an instance-equivalent query Q'. They focus on identifying the selection predicates in select-project-join queries and for generating the selection conditions use a decision tree classifier that is constructed in a top-down manner in a greedy fashion. Zhang et al. [10] compute a generating *join query* that produces a table $Q(D)$ from the tables in D. The generating join query does not have selection conditions and they mostly focus on identifying the joins using graph structures following foreign/primary-key links. Psallidas et al. [8] propose a candidate-enumeration and evaluation framework for discovering project-join queries. Their system handles only text columns and they establish a query relevance score based evaluation of candidate queries. To the best our knowledge, there has not been work that focuses on reverse engineering top-k ranking queries containing aggregation functions.

5. CONCLUSION

We proposed the demonstration of PALEO, a framework for reverse engineering OLAP-style database queries. With three topically diverse datasets and three different demonstration scenarios, we emphasize on highlighting not only the key facets behind the PALEO framework but also aim at making the case for harnessing list-oriented querying and exploration as one easily accessible way to extract the essence of databases.

6. REFERENCES

[1] Database Basketball portal. http://www.databasebasketball.com/

[2] DBLP computer science bibliography. http://dblp.uni-trier.de/

[3] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM J. Discrete Math.*, 17(1), 2003.

[4] Internet Movie Database. http//www.imdb.com/interfaces/

[5] K. Pal and S. Michel. A data mining approach to choosing categorical attributes for ranked list. *EDBT, Poster Track*, 2016.

[6] K. Panev and S. Michel. Reverse engineering top-k database queries with PALEO. *EDBT*, 2016.

[7] K. Panev, E. Milchevski, and S. Michel. Computing similar entity rankings via reverse engineering of top-k database queries. *KEYS Workshop*, 2016.

[8] F. Psallidas, B. Ding, K. Chakrabarti, and S. Chaudhuri. S4: top-k spreadsheet-style search for query discovery. *SIGMOD*, 2015.

[9] Q. T. Tran, C. Chan, and S. Parthasarathy. Query by output. *SIGMOD*, 2009.

[10] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. *SIGMOD*, 2013.