Reverse Engineering Top-k Join Queries

Kiril Panev[1], Nico Weisenauer[2], Sebastian Michel[3]

Abstract:

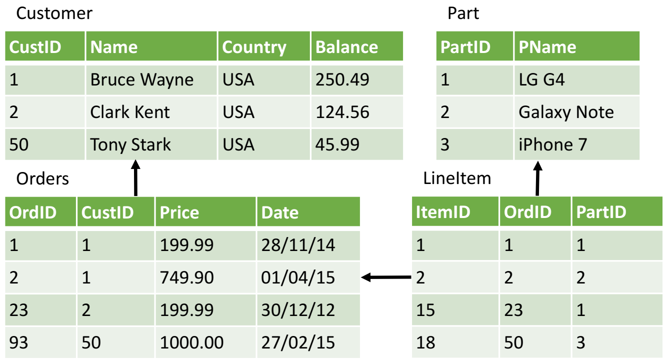
Ranked lists have become a fundamental tool to represent the most important items taken from a large collection of data. Search engines, sports leagues and e-commerce platforms present their results, most successful teams and most popular items in a concise and structured way by making use of ranked lists. This paper introduces the PALEO-J framework which is able to reconstruct top-k database queries, given only the original query output in the form of a ranked list and the database itself. The query to be reverse engineered may contain a wide range of aggregation functions and an arbitrary amount of equality joins, joining several database relations. The challenge of this work is to reconstruct complex queries as fast as possible while operating on large databases and given only the little amount of information provided by the top-k list of entities serving as input. The core contribution is identifying the join predicates in reverse engineering top-k OLAP queries. Furthermore we introduce several optimizations and an advanced classification system to reduce the execution time of the algorithm. Experiments conducted on a large database show the performance of the presented approach and confirm the benefits of our optimizations.

Keywords: data exploration, reverse query processing

1 Introduction

Data in modern relational database systems and specifically in data warehouses is often stored in complex schemas. In order to write analytical queries, database users need to understand the entire schema, which limits the use of the systems to expert users already familiar with the data. Different research areas are working on developing user-friendly ways for querying and exploring data. In this work, we propose an approach of exploring database contents by means of ranked lists. Rankings often capture the essence of the available data, they represent a small subset whose characteristics are important to investigate. In our approach, a user submits a ranked list of entities with their scores and the system returns a set of queries that when executed over the database generate results that match the user's list. We identify OLAP-style select-project-join queries with arbitrary number of joins. There are various important application scenarios. Consider, for instance, business analysts who want to find the different factors that yield some entities of interest as top-performing. By inspecting an identified query, they will learn about the common characteristics of the top entities and about the way the data is organized in the underlying database. Furthermore, more efficient queries wit less-complex join condition can also be found as alternatives. Previous work [Zh13, Sh14, Ps15] considered the problem of reverse engineering join

[1] TU Kaiserslautern, Germany, panev@cs.uni-kl.de
[2] TU Kaiserslautern, Germany, n_weisenau10@cs.uni-kl.de
[3] TU Kaiserslautern, Germany, michel@cs.uni-kl.de



Customer

CustID	Name	Country	Balance
1	Bruce Wayne	USA	250.49
2	Clark Kent	USA	124.56
50	Tony Stark	USA	45.99

Part

PartID	PName
1	LG G4
2	Galaxy Note
3	iPhone 7

Orders

OrdID	CustID	Price	Date
1	1	199.99	28/11/14
2	1	749.90	01/04/15
23	2	199.99	30/12/12
93	50	1000.00	27/02/15

LineItem

ItemID	OrdID	PartID
1	1	1
2	2	2
15	23	1
18	50	3

Fig. 1: Sample database with sales data

Tony Stark	1000.00
Bruce Wayne	749.90
Clark Kent	199.99

(a) Input L

```
SELECT c.Name, MAX(o.Price)
FROM Customer c, Orders o
WHERE c.CustID = o.CustID
AND c.Country = 'USA'
GROUP BY c.NAME
ORDER BY MAX(o.Price) DESC LIMIT 3
```

(b) Result query

Fig. 2: Example input L and result query

queries, however none of them support OLAP-style queries with aggregation functions nor are based on a ranked top-k input.

Figure 1 shows a sample database containing sales data. It contains information about customers, like where they come from, the orders they placed, what parts were bought in each order etc., as in TPC-H [TP]. The arrows denote the direction of foreign key to primary key relationship between the tables. In this scenario it is very common to join the relations to gain insight about the ordering patterns of customers.
Now, consider the top-k list shown in Figure 2a. It contains the customer name and the score that was used as ranking criteria according to which the customers are ordered. Considering the database in Figure 1, we can find that the top-k list can be generated using the query in Figure 2b. It computes the top-3 customers, living in the USA, ranked by the maximum price of an order that they made. There can be other queries that produce the same input list, which enables further exploration of the underlying data. Reverse engineering even this rather simple top-k query on the sample database is not trivial, as the information that can be derived from a two-dimensional top-k list is very limited, especially with small k.

1.1 Problem Statement

Given a database D with tables $\{T_1, T_2, \ldots, T_t\}$ with schema $T_i = \{A_{i_1}, A_{i_2}, \cdots\}$, and an input list L that represents a ranked list of items with the numerical values that were used for the ranking. The system task is to efficiently and effectively discover queries that, when executed over the database D, compute result lists that match the input L.

```
SELECT L.e, agg(value)
FROM R1, R2, ...
WHERE P1 and P2 and ...
GROUP BY L.e
ORDER BY agg(value) DESC
LIMIT k
```

(a)

L	
$L.e$	$L.v$
e	100
f	90
g	80
m	70
o	60

(b)

Fig. 3: Query template (a) and example input L (b)

We identify top-k select-project-join queries of the form shown in Figure 3a. This task can be broken down into three sub-tasks: identifying the join predicates, finding the filter predicates and ranking criteria, and validating the queries. In this work, we specifically focus on finding the query graph, i.e., identifying the foreign key/primary key constraints of the query. We abbreviate primary key as pk, and foreign key as fk in the remainder of this paper.

Definition 1 *A **query graph** is a graph Q with nodes corresponding to database tables in D and edges representing fk/pk constraints.*

The task is to efficiently identify the join constraints of a valid query, i.e., one whose result matches L. This consists of determining the tables in the FROM clause of the query (e.g., Customer c, Orders o), as well as the fk/pk relationships therein (e.g., c.CustID = o.CustID). Identifying filtering predicates and ranking criteria was discussed in prior work [PM16] where the system operates on a single table T and joins were not considered.

The ranked list L, which serves as input for the system and resembles the desired query output, consists of two columns and k rows. The first column, denoted $L.e$, contains entity identifiers, while the second column, denoted $L.v$, contains possibly aggregated numerical values. The input is ordered by the second column, meaning the top-k entities are ranked based on their “score”. L does not contain any information about the database columns containing the entities or the aggregated score values. An example input list is shown in Figure 3b; the column names are omitted in the input as they need not generally be related to database column names.

In addition to finding the fk/pk join constraints, we support predicates P of the form $P_1 \wedge P_2 \ldots \wedge P_m$, where P_i is an atomic equality predicate of the form $A_i = v$ (e.g., Country = 'USA'). We denote with *size* of a predicate $|P|$ the number of atomic predicates P_i in the conjunctive clause.

1.2 Contributions and Outline

In this work, we present an approach that specifically deals with solving the task of reverse engineering top-k *join* queries with arbitrary number of fk/pk joins. With this paper we make the following contributions:

- We present the PALEO-J framework which efficiently computes a list of join query candidates by making use of database meta-data and appropriate data structures.
- Additionally, we present a ranking and verification system which ensures fast execution of candidate join queries. Furthermore, queries with a high probability of being a match are tested first while keeping database interactions low.
- We introduce several optimizations which improve the running time of the framework, including a candidate classification system.
- We present the results of experimental evaluation conducted on a database containing TPC-H [TP] benchmark data.

The paper is organized as follows. Section 2 discusses related work. Section 3 presents system overview. Section 4 establishes the key ideas of our approach to handling joins, followed by introducing certain optimizations in Section 5. Section 6 gives a brief overview of the remaining parts of our framework, while Section 7 reports on the results on the experimental evaluation and presents lessons learned. Section 8 concludes the paper.

2 Related Work

Reverse engineering queries was considered by Tran et al. [TCP09, TCP14] in a data-driven approach which models the problem of finding instance-equivalent queries as a data classification task. This dynamic class-labeling technique is built on a decision tree classifier where nodes are split based on the Gini index, indicating their contribution to the desired query output. While this approach is useful to discover instance-equivalent queries which may be a simpler representation of the original query or help to understand the database schema better, it cannot be used to discover complex OLAP queries given a top-k input list.

Sarma et al. [Sa10] consider only one relation without joins and projections. They focus on finding the selection condition and consider it as an instance of the set cover problem [Va01]. They present separate approaches for different types of queries and we are interested in conjunctive queries with any number of equality predicates. Their proposed algorithms are rather limited and utilize the size of the attribute domains in the view.

Psalidas et al. [Ps15] study the problem of discovering project-join queries which *approximately* contain a given set of example tuples to ease the process of understanding the database schema and writing a specific query. They generate the top-k output list of suitable queries and do not support top-k queries. The systems relies on offline building of large indices and introduce a spreadsheet-style keyword search system based on candidate-enumeration and candidate-evaluation with a flexible caching component. Their scoring model allows to tolerate relationship and domain errors with the given example tuples to make up for human errors while still providing a relevant top-k list of queries.

Shen et al. [Sh14] focus more on the verification of candidate queries and do not consider selection. The system also takes a set of example tuples as input, but tries to discover the minimal project-join query containing the exact set of example tuples in its output. The system requires the database tables to contain only textual data to apply the approach of pruning many invalid candidate queries at once by applying so called *filters*, which resemble the main contribution of their work. The filters exploit subtree relationships between the candidates and rely on full-text-search indices on each text column in the database.

Li et al. [LCM15] propose finding queries from examples, to help non-expert database users construct SQL queries. The user first provides a sample database *D* and an output table *R* which is the result of *Q* on *D*. The proposed approach is able to identify the user's target query by asking for the user's feedback on a sequence of a slightly modified database-result pairs, which are generated by the system. The authors use the work of [TCP09] for generating candidate queries and focus on optimizing the user-feedback interactions to minimize the user's effort in identifying the desired query.

Reverse engineering complex join queries is studied by Zhang et al. [Zh13]. Their main insight is that any graph can be characterized as a union of disjoint paths, connecting its projection tables to a center table, called a *star*, as well as a series of merge steps over this star table. Using this insight, the proposed algorithm filters out candidate queries that need not be tested. To keep track of the candidates, a lattice structure is used where each vertex represents a star and the edges represent merge steps. While this system allows to efficiently discover complex join queries, it is not able to handle top-k OLAP queries containing aggregations or selections. We use their approach as guidance, however using a small top-k list as input and supporting aggregation functions change the problem significantly.
None of these systems allow the reverse engineering of complex top-k OLAP queries including aggregations and fk/pk joins based on a ranked top-k input.

Reverse engineering query processing is studied in [BKL07,Bi07,BCT06,MKZ08], however their objectives and techniques are different. Binning et al. [BKL07, Bi07] discuss the problem of given a query *Q* and a desired result *R*, generate a test database *D* such that, $Q(D) = R$. Bruno et al. [BCT06] and Mishra et al. [MKZ08] consider generating test queries to meet certain cardinality constraints on their subexpressions.

3 System Overview

The system for discovering top-k SPJ queries with arbitrary number of joins contains three main components. We see the three main components depicted in Figure 4.

- Identify the join predicate, i.e., what are the possible fk/pk constraints in the query
- Identify the filter predicates and ranking criteria
- Validate queries

The first component, resembling the contribution of this work, consists of generating a ranked list of candidate joins by using mostly database meta-data and minimal access to actual table data.

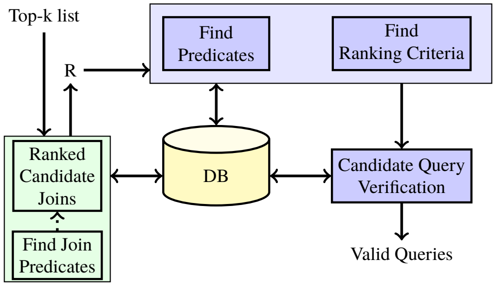


Fig. 4: System architecture

Definition 2 *We say a query graph Q with a set of tables T_i and their fk/pk join predicates is a* ***candidate join*** *iff projecting and selecting on the table R created from the join can result in the input list L. Formally,*

$$R = \bowtie (T_i \in Q), \ \exists \ projection \ \pi, \ selection \ \sigma \ : \ \pi(\sigma(R)) = L$$

The table R created from a candidate join contains all the tuples from the input list L. Furthermore, it contains additional tuples that are not (yet) filtered out by a filter predicate and other columns that will not be used by the projection. Additionally, it contains multiple tuples per entity from which the ranking criterion needs to compute an aggregated score. Thus, the table R is a superset of the input list L and contains all the necessary data to produce the input list by applying the appropriate filter predicates and ranking criterion.

The second component takes as input the candidate join predicates and, for each created join table R, identifies possible filter predicates and ranking criteria. Then, by combining the join conditions, the filter predicates, and the ranking criteria, full-fledged candidate queries are created. The final component verifies these queries by executing them on the database and comparing their results with the input, those that match the input are returned as valid queries. The latter parts of our system are briefly reviewed in Section 6. In this paper, we devise an efficient approach in identifying the join predicates and consider techniques that could facilitate the next steps to determine the remaining parts of a valid query.

4 Finding Join Predicates

The task of identifying candidate join trees consists of the steps depicted in Figure 5. These steps are all part of the Find Join Predicates component, shown in the lower-left part of Figure 4. The following sections will describe the six steps needed to generate the ranked candidate joins from the input L, which will then be processed by the remaining components of the framework.

4.1 Step 1: Schema Exploration

In the Schema Exploration step, the system tries to determine which column in the database schema was used as the entity column. Furthermore, by leveraging basic database metadata, it selects suitable columns as early candidates for the ranking criteria.

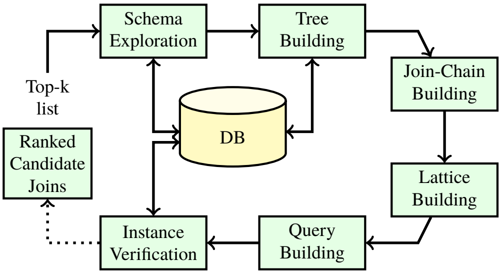


Fig. 5: Generating the ranked candidate joins

Identifying the correct entity column is a straight-forward task. Finding the column from which the input entities originated is done by using a B+ tree index, or alternatively, the entity column can be efficiently identified by implementing an auxiliary inverted index. Finding suitable ranking criteria, however, cannot be done by simple inclusion check of the values in L, since these may stem from an aggregation. Thus, candidate ranking columns are determined based on the data type of the columns.

4.2 Step 2: Building Join-Trees

For each entity candidate column $T.e$, this step explores the database schema and tries to build a tree-like query graph that we call a join tree rooted at $T.e$. All candidate joins are extracted from such trees. We define a depth d of the tree that limits the complexity of the join candidates inspected by the following stages of the framework. A sufficiently high depth d guarantees the successful discovery of a matching query generating L. In turn, if d is chosen too high, the performance of the algorithm may suffer, as a large amount of overly complex candidate joins will be generated.

An example jointree of depth 2 is illustrated in Figure 6 following the TPC-H schema, with nodes corresponding to instances of database tables and edges relating to key-constraints between tables. Starting from the entity candidate table Customer as a root node, the child table Nation is referenced by a fk inside the Customer table, while the Orders child node contains a fk referencing the pk of the Customer relation. All tables related by fk/pk keys will be added as child nodes to the current node, until the maximum depth of the tree has been reached. Hence, each node except for the root node contains its parent node as a child node, unless it is a leaf of the tree. As this results in a lot of nodes instantiating the same database relation, a running index is appended to each node identifier, which now consists of the table name and a number (e.g., Customer1). Having multiple instances of the same table in a join can serve as a proper filter and removing one of them changes the output. Unique node identifiers are important for referencing nodes in later steps of the algorithm, especially when nodes are being merged. Additionally, all of the edges carry information about the table and column names involved in the join to ease the translation into SQL during the Query Building step. The process of building the tree does not have a large performance impact on the database, as the database metadata needs to be queried only once to gain information about all key-constraints in the relevant schema.

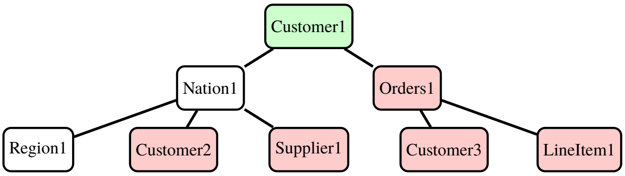


Fig. 6: Example tree of depth 2 with marked nodes

Marking nodes in join trees. After a tree is built, the algorithm continues with marking certain nodes. All ranking candidate columns which have been identified during Schema Exploration will be looked up in the tree and nodes originating from the corresponding tables will be marked as ranking candidate nodes $T.r$, which may also include the root of the tree. Figure 6 shows an example tree with ranking candidate nodes marked in red. The root node of the tree is both an entity and a ranking candidate and therefore marked with green in the figure. Marking the nodes helps to simplify the approach of constructing the join chains, which always have to include at least one marked node and the root of the tree.

4.3 Step 3: Building Join Chains

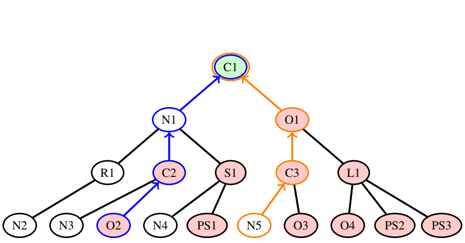
The goal of building the join chains and the lattices during the following step is to represent all possible query graphs connecting the entity table to a table containing a ranking candidate column. Hence a join chain is a set of nodes from the join tree built in Step 2 which includes the entity node and at least one marked node.

Definition 3 *A **join chain** $\mathcal{J}$ is a linear query graph that always contains the database table T_e which was identified as the entity table during Schema Exploration as one node of the graph and at least one node corresponding to a table T_r which contains a ranking candidate column.*
A join chain $\mathcal{J}$ generalizes a query graph Q if the following two conditions are true:

1) *The set of tables corresponding to nodes in Q is equal to the set of tables corresponding to nodes in $\mathcal{J}$*
2) *Each node in Q can be mapped to a node in $\mathcal{J}$ and this mapping is an injective function relating nodes to nodes corresponding to the same database table.*

To construct a join chain, the method begins with a random node in a join tree $\mathcal{T}$, which serves as the starting point of the join chain, and then navigates to its parent node, adding it to the chain in the process. This step is repeated until the root of the tree has been reached, concluding the join chain construction by adding the entity node T_e. By consecutively using all individual nodes contained in the tree as a starting point for a join chain, all possible chains are eventually constructed.

Figure 7a shows an example join tree of depth 3. Two join chains are highlighted in the tree, one starting at the node O2, and the other starting at N5. Both join chains essentially contain the same set of tables: two instances of the Customer relation and one instance each



(a) (b)

Fig. 7: Example join tree (a) and merging of nodes (b)

of the Orders and Nation relation. Hence, these join chains only differ in the order of the join operations, the reason for this being the multiple instances of the Customer relation.

4.4 Step 4: Building Lattices

Join-chains are linear graphs connecting nodes of a tree to its root. The join chains generated during the previous step will serve as root nodes for the lattices built in this next step of our approach. By iteratively merging duplicate nodes, i.e., nodes stemming from the same database table, new query graphs are built, corresponding to vertices inside the lattices, as shown in Figure 8. Each edge in the lattice translates to one merge step. By performing these merge steps over all join chains, all possible query graphs will be built eventually, meaning the query graph representing the query generating L will be among them if the depth of the computation was chosen sufficiently high.

Theorem 1 *If the data inside the database has not changed, then for any query graph Q, there exists at least one join chain $\mathcal{J}$ that generalizes Q, and Q is a node in the lattice rooted at $\mathcal{J}$.*

Theorem 1 is inspired by Section 3 in [Zh13] and its proof follows the one presented there.

Merging nodes. Theorem 1 assures that the query graph of the query which generated L is among the candidate graphs after building the lattices. By merging nodes representing identical database tables, the complexity of the individual graphs decreases with every merge step performed, as the number of nodes inside a graph is reduced.

Unless the depth of the tree that the join chains originated from is very low, there will be a number of nodes referencing identical database tables. This leads to multiple merge steps that might be needed to generate a fully merged graph, as seen in Figure 8. In this situation it is important to maintain the unique identifiers of merged nodes during intermediary merge steps to be able to distinguish nodes, as they could be mapped to the same identifier after performing pairwise merging.

The process of generating a set of lattices starts with adding the root of the lattice and performs a merge operation for each possible pair of nodes referencing identical database

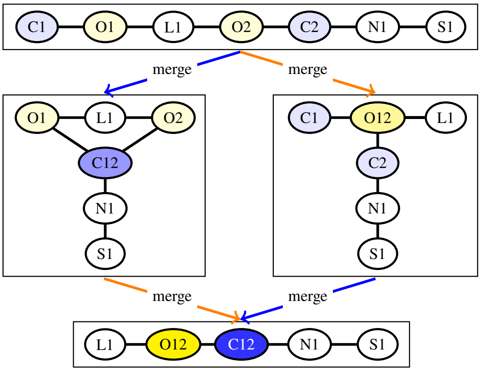


Fig. 8: Example lattice

tables, which will result in a row of new graphs. To successfully merge a pair of two nodes, the resulting merged node retains the union of edges and neighbors of the two original nodes. To indicate that the resulting node is a product of a merge step, it will have a modified identifier which consists of the table name and the two number identifiers of the original nodes appended to each other. After this first iteration, the method continues with the graphs resulting from the first merge step and performs further merging to expand the lattice with new graphs until all possible merge steps have been performed and all graphs have been inspected.

When adding new graphs to the lattice, the lattice keeps track of the number of merge steps needed to generate the newly added graph and keeps graphs with the same amount of merge steps in the same set. This allows us to establish the lattice structure with the initial join chain at the top and the fully merged graph with no nodes referencing identical tables at the bottom.

Handling duplicate graphs. The two join chains visualized in Figure 7a are identical with respect to the set of referenced database tables, and both contain two instances of the Customer relation. Figure 7b shows that both of these join chains collapse to the same chain after performing a merge step on the two Customer nodes. The resulting graph connects the entity candidate Customer relation to the ranking candidate Orders, but also retains the join between the Customer and the Nation relation. The join with the unmarked Nation node appears to not have an impact on the result of the query, as it does not connect the entity table to a ranking candidate, but it may still have an influence on the outcome of a query utilizing predicates to filter Customer entities based on attributes of the Nation relation. Thus, it is important to retain the unmarked nodes, even when they are “dangling” nodes as shown in the example.

After merging the nodes in the two join chains of the example, one of the result graphs can be eliminated due to being a duplicate, therefore reducing the overall number of candidates. Keeping track of duplicate graphs is essential to reduce the computation time of the upcoming steps, which require database interactions and therefore represent a bottleneck of the overall computation.

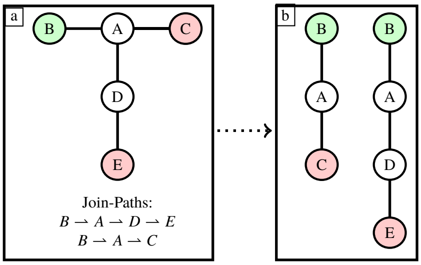

a
B
A
C
D
E
Join-Paths:
$B \rightharpoonup A \rightharpoonup D \rightharpoonup E$
$B \rightharpoonup A \rightharpoonup C$
b
B
A
C
B
A
D
E

Fig. 9: Join path representation of different query graphs

4.5 Step 5: Query Building

Converting a query graph into an SQL query requires extracting various pieces of information to form the SELECT, FROM, and WHERE clause. The query graph stores this information and facilitates the construction of SQL queries.

Each join-query gets assigned a cost value that is used for prioritizing the least costly candidate joins to be tested first. We use the estimated size of the join-result table R as an indicator of how expensive is a candidate join. The relative size of R can be calculated by a simple formula as proposed by Swami and Schiefer [SS94].

$$|T_1 \bowtie T_2| = \frac{|T_1| \times |T_2|}{max(d_1, d_2)} \tag{1}$$

where d_1 and d_2 are the cardinality of the join-columns in T_1 and T_2, respectively.

The queries are then sorted in ascending order of their cost and handed to the next step of the computation, the Instance Verification, which aims for eliminating some of the query candidates before the final candidate list is passed on the next components of the framework.

4.6 Step 6: Instance Verification

During Instance Verification the scores present in the input list L will be compared to actual values in the database columns to rule out candidate joins which do not contain suitable ranking columns. Since the database interactions needed for this step are expensive, it is important to make use of an efficient verification approach which keeps the number of tuples involved in the database operations to a minimum.

Queries may contain joins with tables which are not connecting the entity table to a table containing candidate ranking columns. The goal of Instance Verification is to eliminate candidate ranking columns which cannot generate the input scores. Hence, our goal is to represent each candidate join as one or multiple paths between the table containing the entities and the tables containing the ranking columns.

Figure 9 displays candidate joins represented by their query graphs, the entity table is marked in green while the tables containing ranking candidates are marked in red. The first, shown in Figure 9a, can be represented by the two paths $B \rightharpoonup A \rightharpoonup D \rightharpoonup E$ and $B \rightharpoonup A \rightharpoonup C$.

If a query graph can be represented by two or more paths of various lengths, it usually shares those paths with less complex queries that have already been covered by Instance Verification, as the arriving queries are ordered by their estimated cost. Two less complex queries are shown in Figure 9b, each representing one join path of the query in Figure 9a. By reusing results of already processed queries we can speed up the verification of more complex queries.

To generate the join paths used by Instance Verification, Dijkstra's Shortest Path Algorithm [Co01] is used on the query graphs. Dijkstra's Algorithm will calculate the shortest distance, meaning the least amount of joins needed, from the entity node to each of the ranking nodes, which will result in the respective join paths being built.

The prefix-path relationship. One important assumption about the OLAP-queries generating the input list L is that all joins which are not connecting an entity table to a ranking table do not increase the number of tuples used for score aggregation, hence they are used for applying filters. Otherwise, one would aggregate over duplicate values in the ranking column, resulting from joining the ranking table with another table in pk/fk direction.

Having calculated a join path for each candidate join query, we can use them to reduce the computation time of the actual Instance Verification. This can be achieved by keeping track of the validity of previously verified queries. If a valid candidate query's join path is a prefix of another query's join path, that query can also be considered a valid candidate. This is the case because the less complex query already contains suitable ranking columns, which are also present in the more complex one. This complex query may apply further filtering or not, hence making it a valid candidate query in any case as the boundary conditions for the ranking columns stay the same. In the opposite situation, where a non-valid candidate's join path is a prefix of another query's join path, the algorithm can make use of this result by not testing the already invalidated columns again. For queries which do not share a prefix with an already verified query we can make no assumptions and therefore a full verification needs to be performed.

Candidate join query validation. By default Instance Verification takes the top entity from the input L and evaluates the previously computed join path on the database by executing the respective SQL-query. It is possible to consider more entities from L, resulting in a higher computation time, but also with the possibility of eliminating more ranking candidates.

The result of joining the top entity along the join path is a verification relation V, which contains all possible ranking columns. In order to be a valid ranking candidate, a column has to be able to generate the score in L corresponding to the entity used for generating V. When aggregating the values in a ranking candidate column, all tuples corresponding to the input entity will be used to generate this aggregated score. Therefore this score cannot be directly compared to the score in L, since the aggregation generating the input score may have used only a subset of those tuples, as additional predicates may have filtered some of them out. Hence, score boundaries can only be defined by utilizing the SUM- and MIN-aggregation functions on the columns of this table.

Figure 10a depicts the decision tree showing how a query will be classified as either a valid or non-valid candidate join. The values of the columns being considered are denoted as c_i and c_j. Note that for a query to be valid candidate, only a single column or column pair needs to be classified as valid, whereas for the query to be non-valid, all columns and column pairs need to be classified as non-valid.

The final list of valid candidate queries is ranked by the calculated cost value and handed to the next component of the framework for discovering possible filter predicates and the aggregation function.

5 Optimizations

In this section we propose an advanced classification system to improve the ranking of the candidate join trees, along with other optimizations to decrease the running time of the extended framework.

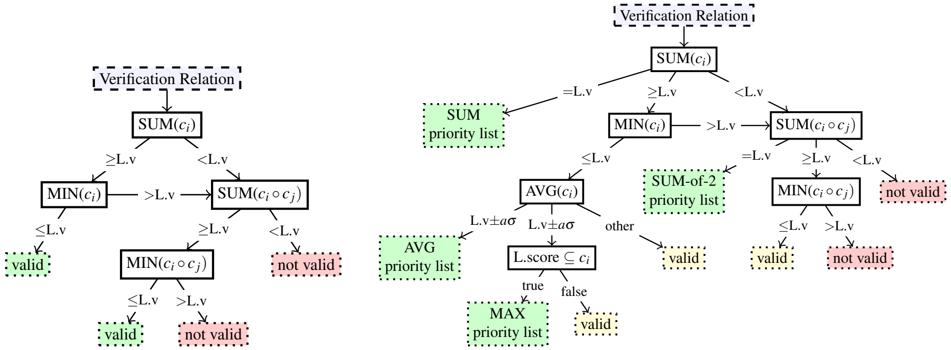
The ranking of candidate join queries has the most influence on the overall running time of the algorithm, as each candidate query has to be executed on the database to compare the actual results to the given input list. In case of long-running queries that contain many cost-intensive joins of large tables, each execution of a candidate query is very time consuming. Our goal is therefore to find the correct query with the first candidates joins that are handed to the rest of the framework, reducing the amount of mismatches to a minimum.

5.1 Improved Query Ranking

The improved query ranking is achieved by identifying possible aggregation functions and columns during the Instance Verification step and by putting the respective queries into higher candidate priority lists. During the identification of the specific aggregation functions, one has to be aware of their particular characteristics which may result in varying precision of identifying the individual functions.

Queries using either the AVG- or the MAX-aggregation function can be identified more precisely than queries using a SUM-aggregation, since the former two aggregations' output values lie within the range of a column's original values. Furthermore, the MAX-aggregation can be identified more reliably than the AVG-aggregation.

After building the priority lists, the candidate join queries inside will be ranked again based on their cost value. The aforementioned lists not only contain the queries which will most likely lead to a matching result query, but also carry information about the aggregation functions which should be considered in the next steps of the framework. To achieve



(a) Basic

(b) Optimized

Fig. 10: Classification of candidate queries

the improved query ranking, more database interactions are needed during the Instance Verification step, which results in a certain overhead being added to every execution. Instead of only considering the sum and the minimum of the values inside a column, now also the mean value and the standard deviation are being computed. This additional information will be used by the algorithm in conjunction with further heuristics to build the priority lists. The improvement in the query ranking outweighs the overhead and we show this in the experiments.

We introduce heuristics that can be applied to build the priority lists, each for the different ranking criteria supported by our system. If the score of an entity lies within a-times the standard deviation of the mean value of the column, then the candidate query containing this column will be entered into a certain priority list. The parameter a should be set sufficiently low, so that few false positives occur, but also high enough to exclude false negatives. Experiments have shown that for the different ranking criteria, a specific margin of the standard deviation fulfills this requirement for most queries.

5.2 Advanced Classification of Queries

The generation of priority lists, and therefore the improved query ranking model, makes use of an advanced classification tree, which is illustrated in Figure 10b. Each of the previously mentioned heuristics for identifying candidate ranking criteria is present inside this classification tree, along with the corresponding priority lists.

A condition containing $L.v$ refers to the score value of the input entity, while $L.score$ refers to a set of score values of the input list L. Note that it is possible to follow multiple edges when checking the conditions in the decision tree, as these are not necessarily mutually exclusive. However, we assume the conditions associated to the edges are checked from left to right and therefore follow a strict order. If the query is not put into any of the various priority lists, it will always be categorized as either *valid* or *not valid* analogous to the baseline Instance Verification approach.

Fast Verification. If a candidate query has been categorized into one of the priority lists, it is very likely to result in a match. Hence the latter parts of the framework should begin processing this query and all related queries as soon as possible. In order to speed up the remaining Instance Verification process, this optimization called Fast Verification allows to skip through the remaining candidates if at least one query has already been put into a priority list.

During Fast Verification, for the remaining candidates, only the join-paths will be inspected. If a join-path of a query inside the priority list is a sub-path of a path of the currently inspected candidate, this candidate will also be put into the same priority list. Otherwise this candidate join query will not undergo any further inspection and will be added to the list of valid join candidates. Thus, if no matching query was found by processing the candidate joins inside the priority lists, such a join query will be handed to the finding filtering predicates component.

6 Discovering Filter Predicates and Ranking Criteria

The techniques for discovering filter predicates and ranking criteria are extensively discussed in [PM16]. The approach there, considers working on a single input table R, which can now be generated in advance by executing the valid candidate joins. As the result of Instance Verification is a ranked list of join-query candidates, they can be executed in order of their rank to generate the table serving as input for the remaining parts of framework, as shown in Figure 4.

To keep the memory usage low, these tables are filtered to only contain the entities from the top-k input list. We denote the joined table that contains only tuples from the input entities as R' and store it in-memory. The system first identifies a set of *candidate predicates* which can be of the form $P_1 \wedge P_2 \ldots \wedge P_m$, where P_i is an atomic equality predicate of the form $A_I = v$ (e.g., country = 'Sweden'). For each candidate predicate there must exist at least one tuple t_i for each entity in L with $P(t_i)$ assessing to true.

A naïve approach would take all possible predicates as candidates and proceed to finding ranking criteria. This would significantly reduce performance, since the expansion of the set of candidate predicates would explode. The system utilizes an apriori-style algorithm that uses the anti-monotone property of the criteria what is considered a candidate predicate. It starts by identifying atomic candidate predicates with size $|P| = 1$ and iteratively creates larger conjunctive predicates by efficiently using the ones created in the previous step.

In order to identify suitable ranking criteria, the system uses the set of determined candidate predicates. It utilizes small data samples, histograms, and simple descriptive statistics which are computed upfront, thus selecting suitable columns and aggregation functions as ranking criteria and avoids touching actual table data. Queries are created by combining the selected ranking criteria with the candidate predicates which are efficiently verified on R'. The queries that produce results that match the input list L are marked as candidate queries, which because of the possible presence of false positives [PM16], need to be verified on the base table that contains *all* tuples, not only tuples with the input entities.

Parameter	Value
depth *d*	5
minJoins	0
maxJoins	3
maxQueries	10

Tab. 1: Parameters used in the experiments

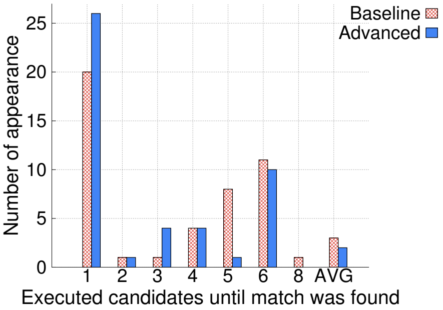


Fig. 11: Number of executed candidate queries

In the last component of our framework, the candidate queries are evaluated using the base table data. Unlike the Instance Verification step presented in Section 4, where the focus is whether the joined tables *are suitable* to produce the input list, here full-fledged candidate queries are executed and their results are compared if they *match* the input list. The queries that match the input list are returned to the user as valid queries.

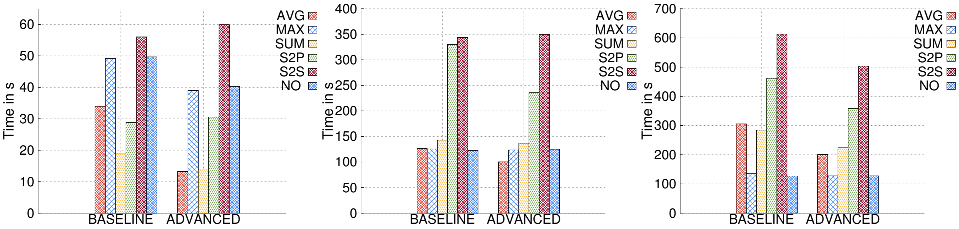
7 Experimental Evaluation

The implementation of our framework was done in Java. The experiments have been conducted on a machine running Ubuntu Linux 14.10, powered by two Intel Xeon E5-2603 hexa-core CPU at 1.60GHz, with 128GB of RAM, using Oracle JVM1.8.0_45 as the Java VM (limited to 20GB memory). The tables are stored in a PostgreSQL 9.4 database, with B+ tree indexes created on the possible entity columns.

Dataset: The evaluation of our system was done using the TPC-H [TP] benchmark. We created a a 10GB dataset, i.e., a scale factor 10 instance of the benchmark.

Input lists: The input used for the experiments consists of 46 query outputs. The queries used to generate the outputs make use of varying scoring functions, between one and three join operations and between zero and three predicates.

All experiments have been conducted with the parameter values shown in Table 1. The depth *d* for generating the tree in Step 2 of the computation has been set to 5, queries with zero to three joins will be considered as candidates and the maximum amount of candidates that will be considered for finding filtering predicates and ranking has been set to 10. These values can be chosen based on the database schema complexity and the estimated complexity of the query to be reverse engineered and allow to limit the execution time of the algorithm by not generating an unnecessary large amount of candidates. Tuning parameter *a* has been set to reasonably small values to allow proper categorization into the priority lists. There were three queries that were outliers, executing their candidate queries on the database took significantly more time than any of the remaining queries in the workload. In order not to skew the results, we do not include these queries in the presented results.



(a) 1-Join (b) 2-Joins (c) 3-Joins

Fig. 12: Average time to find queries with different query graph size $|Q|$

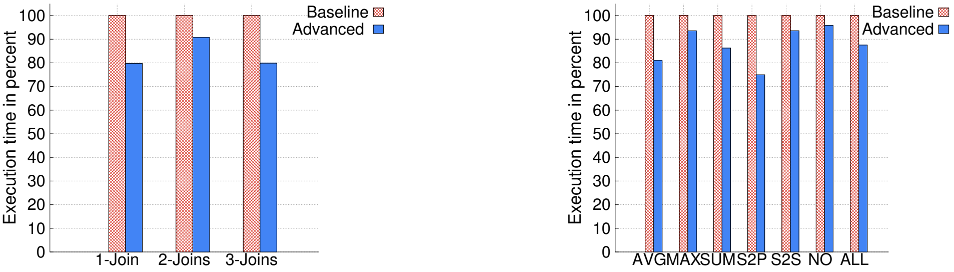
If the basic PALEO-J framework without any optimizations was used to run the experiments, we refer to it as the *baseline approach*, otherwise, we refer to the *advanced approach* when all of the optimizations were utilized. Note that in the evaluation we focus on identifying the join conditions of the query. Finding filtering predicates and ranking criteria is discussed in depth in our previous work. Furthermore, the reported results focus on efficiency of discovering the first valid query in the results that are presented.

Finding a matching query. First, we want to point out that all of the 46 queries have been successfully reverse engineered by the PALEO-J framework, regardless of whether the baseline approach or the advanced approach was applied.

The goal of the optimizations of the advanced approach is to improve the ranking of the candidate join queries, which are given as input to the next steps of the framework. The optimal ranking has the query which is most likely to be a match at the top of the candidate list. Therefore, we show the quality of the ranking by inspecting how many candidate joins have to be executed until a matching query is found.

Figure 11 shows this statistic for the two presented approaches. On average, the advanced approach inspects approximately two queries to find a match while the baseline approach needs about three inspections. The baseline has many cases of inspecting five queries before finding a match and even has a single case where eight inspections were needed. Depending on the time needed to inspect a single candidate query, the difference between inspecting three or two candidate queries on average may be significant, as the following experiments show.

The process of finding a matching query consists of the six steps to generate the candidate list and further finding filtering predicates and ranking criteria by the framework until a matching query is discovered. Figure 12 displays the average time until a matching query is found, grouped by the different scoring functions supported by our framework and the number of joins. While data labeled with AVG, MAX and SUM directly stems from reverse engineering queries using the corresponding aggregation functions, S2P and S2S refers to queries using the *sum of the product* or the *sum of the sum* of two columns. Finally, the label NO corresponds to data stemming from reverse engineering queries with no aggregation function.



(a) By number of joins

(b) By scoring function

Fig. 13: Relative execution times

Figure 13a compares the relative execution times of the advanced approach to the average running time of the baseline as the reference point. On average the advanced approach is faster in reverse engineering a query regardless of the amount of joins and it performs 10–20% faster than the baseline approach.

Figure 13b not only groups the results by scoring function instead of the number of joins, but also averages over the **relative** execution times of reverse engineering individual queries. This is done to make the results between the queries with a different amount of join operators comparable, since they heavily vary in the absolute time needed to find a match. Without this adjustment, the results of reverse engineering queries with three joins would dominate the overall result.

As can be seen in Figure 13b, reverse engineering queries with the MAX-aggregation function or no aggregation function profits the least from the optimizations of the advanced approach, with only about 5% decrease in running time averaged over the amount of joins. The S2P queries profit the most and are discovered about 30% faster. Overall, queries are found almost 14% faster on average when using the advanced approach, ignoring the amount of joins and the used scoring function.

Generating the candidate list. To gain insight into the execution time of the six steps of finding the join predicates, Figure 14 shows the median execution times of these steps based on the 46 test queries.

Steps 1–5 do not vary between the baseline and the advanced approach and also remain relatively constant between the individual queries. Among the first five steps, Step 1 and 2 take the most time, as they are interacting with the database to gain information about the schema. The added up execution times of Steps 3–5 range in the milliseconds even though they are dealing with the more complex data structures, which shows the impact of even the smallest database interactions. The Instance Verification step differentiates the baseline from the advanced approach and the experiments show a significantly increased execution time in the latter, due to the extended classification process. However, this is a small cost to pay compared to the benefit that is gained in the next steps. Looking only at the six steps to generate the list of candidate join queries, we can say that the candidate joins are found within a few seconds for both the baseline and the advanced approach.

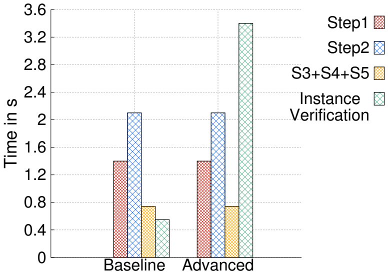


Fig. 14: Execution times of the join-framework steps

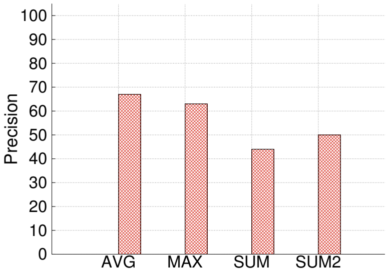


Fig. 15: Precision of categorizing the result query into the correct priority list

Precision of the advanced approach. The improved ranking of the advanced approach can only be achieved if the candidate query which resembles the query generating the input list *L* is categorized into a priority list for early inspection. To gain insight into the accuracy of the advanced classification process, we can check how often candidate queries are put into the priority lists which correspond to the matching queries' scoring function.

Figure 15 shows the precision of this classification grouped by the different priority lists. When comparing this statistic to the one showed in Figure 13b, one can see that the accuracy of the classification does not correlate with the actual performance gain of making use of the improved ranking. This is the case because each aggregation function has a different impact on the quality of the candidate ranking of the baseline approach. The MAX-aggregation seems to result in a good ranking of query candidates when using the regular Instance Verification, while the advanced Instance Verification step invests a lot of time into checking columns for the inclusion of several input scores to put a candidate into a MAX priority list, resulting in a large overhead.

Overall 50% of the scoring functions involving two columns and an average of almost 70% of the scoring functions involving a single column have been correctly identified during Instance Verification of the advanced approach.

Lessons learned. Both the baseline and the advanced approach were able to reverse engineer every single query of the 46 input queries. The advanced approach, which makes use of the advanced classification system, executed fewer candidates before it found the matching query. The advanced classification system also comes with an overhead. This overhead pays off, since the performance gain from improving the ranking of the candidate list depends on the time needed to execute the individual candidates, which are expensive because of the present joins.

The advanced approach always outperforms the baseline and more complex join queries have bigger performance gain. Having more join operations leads to slower candidate query execution, thus the ranking played a bigger role when reverse engineering these complex queries. The advanced approach reduced the time needed for reverse engineering join queries by up to 20% compared to the baseline approach.

8 Conclusion

We presented the PALEO-J framework for reverse engineering top-k database join queries. This is a challenging problem, given the small input and the amount of database tables that have to be considered as potential join partners due to the lack of information to filter candidates out. Having only entities and aggregated scores to find and filter out candidates, we had to find a way to interact with the database system to efficiently classify the large amount of candidate queries. This was achieved with the proposed Instance Verification step, which was refined with the advanced classification system to significantly improve the query ranking. The introduction of priority lists improves the ranking of candidate join queries, which ensures that less false positives will be considered by the remaining parts of the framework.

References

[BCT06] N. Bruno, S. Chaudhuri, and D. Thomas. Generating queries with cardinality constraints for DBMS testing. *IEEE Trans. Knowl. Data Eng.*, 2006.

[Bi07] C. Binnig, D. Kossmann, E. Lo, and M.T. Özsu,: QAGen. generating query-aware test databases. *SIGMOD*, 2007.

[BKL07] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. *ICDE*, 2007.

[Co01] T.H. Cormen, C. Stein, R.L. Rivest, and C.E. Leiserson. Introduction to algorithms. *McGraw-Hill Higher Education*, 2001.

[LCM15] H. Li, C. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. *PVLDB*, 8(13), 2015.

[MKZ08] C. Mishra, N. Koudas, and C. Zuzarte. Generating targeted queries for database testing. *SIGMOD*, 2008.

[PM16] K. Panev and S. Michel. Reverse engineering top-k database queries with PALEO. *EDBT*, 2016.

[Ps15] F. Psallidas, B. Ding, K. Chakrabarti, and S. Chaudhuri. S4: Top-k spreadsheet-style search for query discovery. *SIGMOD*, 2015.

[Sa10] A.D. Sarma, A.G. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. *ICDT*, 2010.

[Sh14] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. *SIGMOD*, 2014.

[SS94] A. Swami and K.B. Schiefer. On the estimation of join result sizes. *EDBT*, 1994.

[TCP09] Q. T. Tran, C. Chan, and S. Parthasarathy. Query by output. *SIGMOD*, 2009.

[TCP14] Q. T. Tran, C. Chan, and S. Parthasarathy. Query reverse engineering. *VLDBJ*, 23(5), 2014.

[TP] TPC. TPC benchmarks. http://www.tpc.org/.

[Va01] V.V. Vazirani. Approximation algorithms. *Springer-Verlag New York, Inc.*, 2001.

[Zh13] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. *SIGMOD*, 2013.