Efficient Query Reverse Engineering using Table Fragments

Meiying Li and Chee-Yong Chan

National University of Singapore
{meiying,chancy}@comp.nus.edu.sg

Abstract. Given an output table T that is the result of some unknown query on a database D, Query Reverse Engineering (QRE) computes one or more target query Q such that the result of Q on D is T. A fundamental challenge in QRE is how to efficiently compute target queries given its large search space. In this paper, we focus on the QRE problem for PJ^+ queries, which is a more expressive class of queries than project-join queries by supporting antijoins as well as inner joins. To enhance efficiency, we propose a novel query-centric approach consisting of table partitioning, precomputation, and indexing techniques. Our experimental study demonstrates that our approach significantly outperforms the state-of-the-art solution by an average improvement factor of 120.

Keywords: Query Reverse Engineering · Query Processing

1 Introduction

Query Reverse Engineering (QRE) is a useful query processing technique that has diverse applications including data analysis [18, 4], query discovery [20, 15, 14, 3, 19], query construction [9], and generating query explanations [17, 6, 7, 5, 11]. The fundamental QRE problem can be stated as follows: given an output table T that is the result of some unknown query on a database D, find a target query Q such that the result of Q on D (denoted by $Q(D)$) is equal T.

Research on the QRE problem can be categorized based on two orthognal dimensions. The first dimension relates to the class of queries being reverse-engineered, and the different query fragments that have been investigated for the QRE problem include select-project-join (SPJ) queries [19, 18, 17, 9], top-k queries [11, 14, 13], aggregation queries [18, 16], and queries with quantifiers [1]. The second dimension concerns the relationship between the target query result $Q(D)$ and the given output table T. One variant of the QRE problem [20, 8, 18, 16] requires finding *exact target queries* such that $Q(D) = T$. Another variant of the QRE problem [15, 14, 9] relaxes the requirement to finding *approximate target queries* such that $Q(D) \supseteq T$. Solutions designed for the first problem variant could be adapted for the second easier problem variant.

In this paper, we focus on the exact target query variant of the QRE problem for the class of Project-Join (PJ) queries, which is a fundamental query fragment [20, 15, 14, 3]. Specifically, we address two key challenges that arise from the

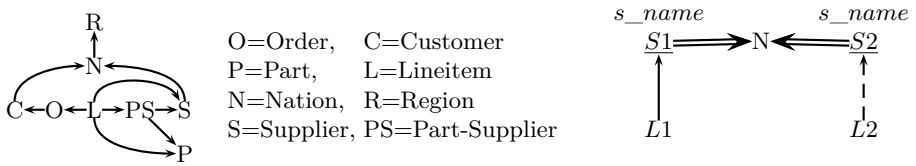
efficiency-expressiveness tradeoff for this QRE problem. The first challenge is the issue of search efficiency; i.e., how to efficiently find a target query from the large search space of candidate queries. Note that the QRE problem is NP-hard even for the simple class of PJ queries [2]. All the existing solutions for QRE are based on a *generate-and-test paradigm* that enumerates the space of candidate queries and validates each candidate query Q to check if the result of Q on D is T. All the existing approaches [20, 8] generate a candidate query in two main steps. The first step enumerates the candidate projection tables for the query (i.e., the tables from which the output columns are projected), and the second step enumerates the candidate join paths to connect the projection tables. We refer to these approaches as *path-centric approaches*.

To address the efficiency of searching for candidate queries, two main strategies have been explored. The first strategy focuses on the design of heuristics to efficiently prune the search space [20, 8]. The second strategy is to reduce the search space by considering only a restricted class of PJ queries; for example, the recent work on `FastQRE` considers only *Covering PJ* (CPJ) queries [8] which is a strict subset of PJ queries. However, even with the use of pruning heuristics, the performance of the state-of-the-art approach for PJ queries, `STAR`, could be slow for some queries; for example, `STAR` took 700s to reverse engineer the target query Q3 in [20][1].

The second challenge is the issue of supporting more expressive PJ queries. All existing approaches support PJ/CPJ queries without negation; i.e., the join operators in the PJ/CPJ queries are limited to inner joins and do not support antijoins. Thus, more expressive PJ queries such as "find all customers who have not placed any orders" can not be reversed engineered by existing approaches. Supporting antijoins in PJ queries is non-trivial as a straightforward extension of the generate-and-test paradigm to simply enumerate different combinations of join operators for each candidate PJ query considered could result in an exponential increase in the number of candidate queries. Indeed, the most recent QRE approach for PJ queries [8] actually sacrifices the support for expressive PJ queries to improve search efficiency over STAR [20].

Thus, the problem of whether it is possible to efficiently reverse engineer more expressive PJ queries remains an open question. In this paper, we address this problem by presenting a novel approach named `QREF` (for QRE using Fragments). Our approach's key idea is based on horizontally partitioning each database table into a disjoint set of fragments. using a set of PJ^+ queries. A *PJ^+ query* is a more general form of PJ queries that supports antijoins in addition to inner joins. By partitioning each table into fragments such that each fragment is associated with a PJ^+ query, `QREF` has the three advantages. First, in contrast to the existing path-centric approaches for generating candidate queries, `QREF` is a *query-centric approach* where given a set of candidate projection tables, it enumerates a set of candidate projection table queries (which are PJ^+ queries) for each projection table, and a candidate query is formed by merging a set of projection table queries. By judicious precomputation of the PJ^+ queries associated with subsets

[1] In contrast, our approach took 3s to reverse engineer this query (Section 6).



(a) Schema Graph G_D (b) Query Graph G_{Q_1}

Fig. 1: (a) Schema graph G_D for TPC-H database D with relation names abbreviated. (b) Query graph for query $Q_1 = \pi_{S1.s\_name,S2.s\_name}((S1 \ltimes L1) \bowtie N \bowtie (S2 \triangleright L2))$.

of table fragments, `QREF` is able to efficiently enumerate candidate projection table queries without the need to enumerate candidate join paths. Second, by indexing the PJ$^+$ queries associated with the table fragments, `QREF` is able to efficiently validate a candidate query Q by (1) accessing only relevant fragments instead of entire relations, and (2) eliminating some of the join computations in Q that are already precomputed by the table fragments. Third, `QREF` is able to reverse-engineer PJ$^+$ queries, which is a larger and more expressive class of queries than the PJ/CPJ queries supported by existing approaches [20, 8].

In summary, this paper proposes a novel query-centric approach to solve the QRE problem for the target class of PJ$^+$ queries based on table partitioning, precomputation, and indexing techniques. Although we have developed these techniques specifically for the class of PJ queries considered in this paper, our techniques could be adapted to optimize the QRE problem for other query fragments as well. To the best of our knowledge, our work is the first to reverse engineer queries with negation (specifically queries with antijoins).

2 Background

2.1 Query Classes

Given a database D consisting of a set of relations $\{R_1, \cdots, R_n\}$, we define its **schema graph** to be a directed graph $G_D = (V, E)$, where each node in $v_i \in V$ represents relation R_i, and each directed edge $(v_i, v_j) \in E$ represents a foreign-key relationship from R_i to R_j. In case if there are multiple foreign-key relationships from a relation to another relation, the directed edges would also be labeled with the names of the pair of foreign-key and primary-key attribute names. Fig. 1a shows the schema graph for the TPC-H schema.

We now define all the fragments of PJ queries that have been studied in the past (i.e., PJ, PJ$^-$, CPJ) as well as the most general fragment that is the focus of this paper (i.e., PJ$^+$) These four fragments are related by the following strict containment relationship: $CPJ \subset \text{PJ}^- \subset PJ \subset \text{PJ}^+$.

A **Project-Join (PJ) query** is of the form $\pi_\ell(E)$, where E denote a join expression involving innerjoins ($\bowtie$), with all the join predicates based on foreign-

key joins[2], and ℓ denote a list of attributes. A **Project-Join$^+$ (PJ$^+$) query** is a more general form of PJ queries that also supports left-antijoins ($\triangleright$) in addition to innerjoins (and left-semijoins ($\ltimes$)). Thus, the class of PJ$^+$ queries are strictly more expressive than PJ queries as every PJ query is also a PJ$^+$ query.

A PJ$^+$ query Q can be represented by a **query graph** G_Q, where each relation in Q is represented by a node in G_Q, and each join between a pair of relations R_i and R_j in Q is represented by an edge between their corresponding nodes as follows: a innerjoin, left-semijoin, and left-antijoin, are denoted respectively, by a double-line, single-line, and dashed-line edges. The edges between nodes are directed following the edge directions in the schema graph.

To distinguish multiple instances of the same relation, we append a unique number to the relation instance names for convenience. If the projection list in Q includes some attribute from a relation R_i, then we refer to R_i (resp. the node representing R_i) as a **projection table** (resp. **projection node**). Each projection node in G_Q is underlined and annotated with a list of its projection attributes.

The non-projection nodes in a query graph are further classified into connection nodes and filtering nodes. Given a path of nodes $p = (v_1, \cdots, v_m)$ in G_Q, where the end nodes v_1 and v_m in p are the only two projection nodes in p, each $v_i, i \in (1, m)$ is a **connection node**. A node that is neither a projection nor a connection node is a **filtering node**.

A key feature of our approach is to horizontally partition each database table using a subclass of PJ$^+$ queries that we refer to as **filtering PJ$^+$ queries (FPJ$^+$)**. A FPJ$^+$ query is a PJ$^+$ query that has exactly one projection table and using only left-semijoins and/or left-antijoins. Thus, each FPJ$^+$ query consists of exactly one projection node and possibly some filtering nodes without any connection node.

A **Project-Join$^-$ (PJ$^-$) query** is a restricted form of PJ query that does not contain any filtering nodes. A **Covering Project Join (CPJ) query** is a restricted form of PJ$^-$ query with the constraint that for any two connection nodes v_i and v_j that correspond to two instances of the same table, v_i and v_j must be along the same path that connects a pair of projection nodes. Given a query Q, we define its size, denoted by $|Q|$, to be the number of nodes in its query graph.

Example 1. Fig. 1b shows the query graph G_{Q_1} for the PJ$^+$ query Q_1 (on the TPC-H schema) that returns the names of all pairs of suppliers ($sname_1$, $sname_2$) who are located in the same country, where $sname_1$ has supplied some lineitem, and $sname_2$ has not supplied any lineitem. In G_{Q_1}, $L1$ and $L2$ are filtering nodes, N is a connection node, $S1$ and $S2$ are projection nodes, and $|Q_1| = 5$. If N and its incident edges are removed from G_{Q_1}, then we end up with two FPJ$^+$ queries $\underline{S1} \ltimes L1$ and $\underline{S2} \triangleright L2$. Note that Q_1 is not a PJ query

[2] Although our experiments focus on queries with foreign-key joins (similar to all competing approaches [20, 8]), our approach can be easily extended to reverse engineer PJ queries with non-foreign key join predicates. The main extension is to explicitly annotate the database schema graph with additional join edges.

due to the antijoin $\underline{S2} \triangleright L2$; if the antijoin in Q_1 is replaced with a semijoin (i.e., $\underline{S2} \ltimes L2$), then the revised query Q'_1 becomes a PJ query as the semijoins in Q'_1 are effectively equivalent to innerjoins. Note that Q'_1 is not a CPJ query due to the filtering nodes $L1$ and $L2$; if these filtering nodes are removed from Q'_1, then the resultant query is a CPJ query. □

3 Our Approach

In order to address the efficiency challenge of supporting PJ$^+$ queries, which is a even more expressive class of queries supported by `STAR` [20], we design a new approach, termed `QREF`, that applies table partitioning, precomputation and indexing techniques to improve the performance of the QRE problem. To the best of our knowledge, our approach is the first to utilize such techniques for the QRE problem and our work is also the first to support negation (in the form of antijoins) in target queries.

Query-centric Approach. In contrast to existing path-centric QRE solutions, `QREF` is a *query-centric* approach that treats each PJ$^+$ query as a combination of FPJ$^+$ queries. Referring to the example target PJ$^+$ query Q_1 in Fig. 1b, Q_1 can be seen as a combination of two FPJ$^+$ queries, $Q_{S_1} = \pi_{s\_name}(\underline{S} \ltimes L \ltimes N)$ and $Q_{S_2} = \pi_{s\_name}(\underline{S} \triangleright L \ltimes N)$. The target query Q_1 can be derived by merging these two FPJ$^+$ queries via the common table $Nation$.

We illustrate our approach using a running example of a simplified TPC-H database D consisting of the four tables shown in Figs. 2a to 2d; for convenience, we have indicated a *tid* column in each database table to show the tuple identifiers. In this example, our goal is to try to reverse engineer a PJ$^+$ query from the output table T shown in Fig. 2e.

Candidate Projection Tables. The first step in `QREF`, which is a necessary step in all existing approaches, is to enumerate possible *candidate projection tables* that could be joined to compute the output table T. Intuitively, for a set PT of table instances to be candidate projection tables for computing T, each of the columns in T must at least be contained in some table column in PT; otherwise, there will be some tuple in T that cannot be derived from PT. In our running example, since $\pi_{pkey,brand}(P) \supseteq \pi_{C_1,C_2}(T)$ and $\pi_{name}(S) \supseteq \pi_{C_3}(T)$, one possible set of candidate projection tables to compute T is $PT_1 =$ {P(pkey,brand), S(name)}. We say that *P(pkey,brand) covers* $T(C_1, C_2)$, and P.pkey (P.brand, resp.) is a *covering attribute* of $T.C_1$ ($T.C_2$, resp.). Another set of candidate projection tables is $PT_2 =$ {P(pkey), P(brand), S(name)} which differs from PT_1 in that two instances of the table P are used to derive the output columns C_1 and C_2 separately. Fig. 2k shows the covering attributes for each of the three columns in T; for example, there are two covering attributes (i.e., *P.pkey* and *PS.pkey*) for $T.C_1$. Fig. 2l shows all the possible sets of candidate projection tables for T.

Candidate Queries. Given a set of candidate projection tables PT, the second step in `QREF` is to enumerate candidate queries that join the projection tables in PT. For example, given $PT_1 =$ {P(pkey,brand), S(name)}, one possibility

tid	skey	name
t_1	1	$supp1$
t_2	2	$supp2$

(a) **Table S**

tid	pkey	skey	okey
t_3	2	2	1
t_4	2	2	2

(b) **Table L**

tid	pkey	brand
t_5	1	b_1
t_6	2	b_2
t_7	3	b_2

(c) **Table P**

tid	pkey	skey
t_8	1	1
t_9	2	1
t_{10}	3	1
t_{11}	2	2

(d) **Table PS**

C_1	C_2	C_3
1	b_1	$supp1$
3	b_2	$supp1$

(e) **Output Table** T

Q_{P_1}	$\underline{P} \ltimes L \triangleright PS$
Q_{P_2}	$\underline{P} \ltimes L \ltimes (PS \ltimes L) \ltimes (PS \triangleright L)$
Q_{P_3}	$\underline{P} \ltimes L \ltimes (PS \ltimes L) \triangleright (PS \triangleright L)$
Q_{P_4}	$\underline{P} \ltimes L \triangleright (PS \ltimes L) \ltimes (PS \triangleright L)$
Q_{P_5}	$\underline{P} \triangleright L \triangleright PS$
Q_{P_6}	$\underline{P} \triangleright L \ltimes (PS \ltimes L) \ltimes (PS \triangleright L)$
Q_{P_7}	$\underline{P} \triangleright L \ltimes (PS \ltimes L) \triangleright (PS \triangleright L)$
Q_{P_8}	$\underline{P} \triangleright L \triangleright (PS \ltimes L) \ltimes (PS \triangleright L)$

(f) **Fragment queries to partition** P

Q_{P_2}	$\underline{P} \ltimes L \ltimes (PS \ltimes L) \ltimes (PS \triangleright L)$
SQ_1	$\underline{P} \ltimes L \ltimes (PS \ltimes L)$
SQ_2	$\underline{P} \ltimes (PS \triangleright L) \ltimes (PS \ltimes L)$
SQ_3	$\underline{P} \ltimes L \ltimes (PS \triangleright L)$
SQ_4	$\underline{P} \ltimes (PS \ltimes L)$
SQ_5	$\underline{P} \ltimes L \ltimes PS$
SQ_6	$\underline{P} \ltimes L$

(g) **Query**($\{P_2\}$)

SQ_{13}	$\underline{P} \ltimes (PS \triangleright L)$
SQ_{14}	$\underline{P} \ltimes PS$
SQ_{15}	$\underline{P}$

(h) **Query**($\{P_2, P_8\}$)

Q_{P_8}	$\underline{P} \triangleright L \triangleright (PS \ltimes L) \ltimes (PS \triangleright L)$
SQ_7	$\underline{P} \ltimes (PS \triangleright L) \triangleright (PS \ltimes L)$
SQ_8	$\underline{P} \triangleright L \triangleright (PS \ltimes L)$
SQ_9	$\underline{P} \triangleright L \ltimes (PS \triangleright L)$
SQ_{10}	$\underline{P} \triangleright L \ltimes PS$
SQ_{11}	$\underline{P} \triangleright (PS \ltimes L)$
SQ_{12}	$\underline{P} \triangleright L$

(i) **Query**($\{P_8\}$)

F	$ExtQuery(F)$
$\{P_2\}$	SQ_4, SQ_6, Q_{P_2}
$\{P_8\}$	$SQ_{11}, SQ_{12}, Q_{P_8}$
$\{P_2, P_8\}$	SQ_{13}, SQ_{15}

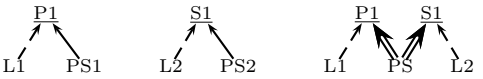
(j) $TIndex\{P\}$.

C_1	C_2	C_3
P.pkey	P.brand	S.name
PS.pkey		

(k) **Covering Attributes for** T

$PT_1 = \{P(pkey, brand), S(name)\}$
$PT_2 = \{P(pkey), P(brand), S(name)\}$
$PT_3 = \{PS(pkey), P(brand), S(name)\}$

(l) **Candidate Projection Tables**



(m) **Query** Q_1 (n) **Query** Q_2 (o) **Merge** $\{PS1, PS2\}$

Fig. 2: Running Example: Simplified TPC-H Database (Tables S, L, P & PS) with $\alpha = 2$

is to join the two projection tables via the intermediate table L to produce the candidate query Q_1: $\pi_{pkey,brand,name}(\underline{P} \bowtie_{pkey} L \bowtie_{skey} \underline{S})$. To avoid the performance issues with the path-centric approach, `QREF` uses a novel and more efficient *query-centric approach* to enumerate candidate queries that is based on a combination of three key techniques: table partitioning, precomputation, and indexing.

Table Fragments & Fragment Queries. `QREF` horizontally partitions each database table into a set of pairwise-disjoint fragments using a set of FPJ$^+$ queries. As an example, Fig. 2f shows a set of 8 FPJ$^+$ queries $\{Q_{P_1}, \cdots, Q_{P_8}\}$ to partition the *Part* table into 8 *table fragments*: $P = P_1 \cup \cdots \cup P_8$. Each table fragment P_i is the result of the FPJ$^+$ query Q_{P_i}, and we refer to each such query as a *fragment query*. In our running example, all the fragments of P are empty except for $P_2 = \{t_6\}$ and $P_8 = \{t_5, t_7\}$. To control the partitioning granularity, `QREF` uses a *query height parameter*, denoted by α, which determines the maximum height of each fragment query graph. Observe that the set of fragment queries in Fig. 2f are all the possible FPJ$^+$ queries w.r.t. P with a maximum height of 2.

Projection Table Queries. Consider a candidate projection table $R(A)$ that covers $T(A')$, where A and A' are subsets of attributes of R and T, respectively; i.e., $\pi_A(R) \supseteq \pi_{A'}(T)$. Instead of enumerating join paths w.r.t. R, `QREF` enumerates FPJ$^+$ queries Q w.r.t. R that are guaranteed to cover $T(A')$; i.e.,

if $R' \subseteq R$ is the output of Q on R, then $R'(A)$ covers $T(A')$. We refer to such FPJ$^+$ queries as *projection table queries*. Thus, unlike the path-centric approach where the enumerated join paths could be false positives, the projection table queries enumerated by `QREF` has the nice property that they cover the output table which enables `QREF` to prune away many candidate queries that are false positives. To provide this pruning capability, `QREF` utilizes a space-efficient *table fragment index* that precomputes the set of projection table queries for each subset of table fragments (Section 4.3).

Consider again the set of candidate projection tables $PT_1 = \{$P(pkey,brand), S(name)$\}$. Fig. 2m (resp. Fig. 2n) shows a candidate projection table query for P (resp. S). Given these candidate projection table queries, `QREF` enumerates different candidate queries by varying different ways to merge these queries. One possibility is to merge the $PS1$ node in Fig. 2m with the $PS2$ node in Fig. 2n to generate the candidate query in Fig. 2o.

Validation of Candidate Queries. For each enumerated candidate query Q, the third step in `QREF` is to validate Q; i.e., determine whether Q computes T. Consider the candidate query Q shown in Fig. 2o. Instead of computing Q directly, which requires a 5-table join, `QREF` is able to optimize the evaluation of Q by making use of the table fragments. Specifically, Q is evaluated as $Q' = \underline{P'} \bowtie PS' \bowtie \underline{S'}$ where P' refers to the table fragment $\underline{P} \ltimes PS \triangleright L$, PS' refers to the table fragment $\underline{PS} \ltimes (P \triangleright L) \ltimes (S \triangleright L)$, and S' refers to table fragment $\underline{S} \ltimes PS \triangleright L$. To efficiently identify relevant table fragments for query evaluation, `QREF` builds an index on the fragment queries associated with the fragments of each table (Section 4.2).

Overall Approach. Our query-centric approach consists of an offline phase and an online phase. The offline phase partitions the database tables into fragments, precomputes candidate projection table queries from the table fragments, and build indexes on the table fragments and fragment queries. The online phase processes QRE requests at runtime, where each QRE request consists of an output table to be reverse-engineered.

4 Table Fragments & Indexes

In this section, we present `QREF`'s offline phase which partitions each database table into fragments, precomputes candidate projection table queries, and builds indexes on the table fragments and query fragments to enable efficient query reverse engineering.

4.1 Partitioning Tables into Fragments

In this section, we explain how each database table is horizontally partitioned into fragments using a set of FPJ$^+$ queries. To control the granularity of the partitioning, `QREF` uses a **query height parameter**, denoted by α, so that the height of each fragment query graph is at most α. Thus, partitioning a table with a larger value of α will result in more table fragments.

We use $Frag(R) = \{R_1, \cdots, R_m\}$ to denote the set of m fragments created by the partitioning of a table R, and use Q_{R_i} to denote the fragment query that computes R_i from R. Each of the tables in the database is partitioned iteratively using FPJ$^+$ queries with increasing query height from 1 to α based on the join relationships among the tables. which are specified by the join edges in the database schema graph.

To make the discussion concrete, we consider the partitioning of three tables PS, P, and L from the schema graph G_D shown Fig. 1(a). Recall that we consider only foreign-key joins in G_D for simplicity, but the approach described can be generalized to other types of joins so long as they are captured in G_D. Each table R in G_D will be partitioned based on the set of foreign-key relationships to R. For the tables $\{PS, P, L\}$ being considered, the foreign-key join relationships are $\{PS \leftarrow L, P \leftarrow L, P \leftarrow PS\}$. In the first iteration, $PS = PS_1 \cup PS_2$ is partitioned into two fragments with the following FPJ$^+$ queries: $Q_{PS_1} = PS \ltimes L$ and $Q_{PS_2} = PS \triangleright L$. Similarly, $P = P_1 \cup P_2 \cup P_3 \cup P_4$ is partitioned into four fragments with the following FPJ$^+$ queries: $Q_{P_1} = P \ltimes L \ltimes PS$, $Q_{P_2} = P \ltimes L \triangleright PS$, $Q_{P_3} = P \triangleright L \ltimes PS$, and $Q_{P_4} = P \triangleright L \triangleright PS$. Since there is no foreign-key relationship into L, L will not be partitioned at all. In the second iteration, the fragments PS_1 and PS_2 will not be further partitioned since L was not partitioned in the first iteration. On the other hand, since PS was partitioned into PS_1 and PS_2 in the first iteration, these will be used to further partition $P_1 = P_{1,1} \cup P_{1,2} \cup P_{1,3}$ into three fragments using the following FPJ$^+$ queries:

$$Q_{P_{1,1}} = P \ltimes L \ltimes PS_1 \ltimes PS_2 = P \ltimes L \ltimes (PS \ltimes L) \ltimes (PS \triangleright L)$$
$$Q_{P_{1,2}} = P \ltimes L \ltimes PS_1 \triangleright PS_2 = P \ltimes L \ltimes (PS \ltimes L) \triangleright (PS \triangleright L)$$
$$Q_{P_{1,3}} = P \ltimes L \triangleright PS_1 \ltimes PS_2 = P \ltimes L \triangleright (PS \ltimes L) \ltimes (PS \triangleright L)$$

Similarly, fragment P_3 will be further partitioned into three fragments using PS_1 and PS_2. However, the fragments P_2 and P_4 will not be further partitioned because the tuples in them do not join with any tuple from PS. Thus, if $\alpha = 2$, the relation P will be partitioned by `QREF` into a total of 8 fragments as shown in Fig. 2f.

The partitioning of tables into fragments using FPJ$^+$ queries with a height of at most α can be performed efficiently by applying α-bisimulation techniques [12] on the database. Furthermore, the techniques from [12] can be applied to efficiently maintain the database fragments when the database is updated.

4.2 Fragment Query Index

For each table R, `QREF` builds a **fragment query index**, denoted by $QIndex(R)$, to index all the fragment queries used for partitioning R. These indexes are used for the efficient validation of candidate queries (to be explained in Section 5.3). Specifically, given an input FPJ$^+$ query Q w.r.t. table R, the fragment query index $QIndex(R)$ will return a list of the identifiers of all the fragments of R that match Q; i.e., the result of the query Q on the database is the union of all the fragments of R that are returned by the index search on $QIndex(R)$ w.r.t. Q.

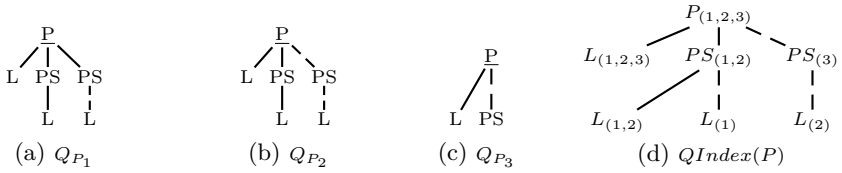


Fig. 3: Fragment Queries & Fragment Query Index $QIndex(P)$ for relation P

Each $QIndex(R)$ is a trie index that indexes all the root-to-leaf paths in the fragment query graphs of R. As an example, Fig. 3d shows $QIndex(P)$, the fragment query index for a relation P, which has been partitioned into a set of three non-empty fragments $\{P_1, P_2, P_3\}$ with their fragment queries shown in Figs. 3a, 3b, and 3c[3]. Each node in $QIndex(P)$ represents a relation, with P as the root node of $QIndex(P)$. Similiar to the edges in a query graph, each edge in $QIndex(P)$ represents either a semijoin (sold line) or an antijoin (dashed line). Each node N in $QIndex(P)$ stores a list of fragment identifiers (indicated by the node's subscript in Fig. 3d) identifying the subset of fragments of $Frag(P)$ that match the PJ$^+$ query represented by the single-path in $QIndex(P)$ that starts from the root node of $QIndex(P)$ to N. As an example, for the second child node PS of P in $QIndex(P)$, its list of fragment identifiers is $(1, 2)$ indicates that only fragments P_1 and P_2 (but not P_3) match the PJ$^+$ query $\underline{P} \ltimes PS$.

4.3 Table Fragment Index

The second type of index built by `QREF` for each table R is the **table fragment index**, denoted by $TIndex(R)$. For each non-empty subset of fragments of R, $F \subseteq Frag(R)$, $TIndex(R)$ stores a set of FPJ$^+$ queries (with a maximum height of α) that computes F from R. We denote this set of queries by $Query(F)$. The table fragment index for R, $TIndex(R)$, is used to efficiently enumerate the set of candidate projection table queries for a projection table R (to be discussed in Section 5.2).

Given $F \subseteq Frag(R)$, we now explain how to derive $Query(F)$. The derivation algorithm uses the the following three rewriting rules (S1 to S3) and additional knowledge of whether any fragments in F are empty. In the following rules, each E_i denote some FPJ$^+$ query.

S1. $(E_1 \ltimes E_2) \cup (E_1 \triangleright E_2) = E_1$
S2. $(E_2 \circ E_1) \cup (E_3 \circ E_1) = (E_2 \cup E_3) \circ E_1$, $\circ \in \{\ltimes, \triangleright\}$
S3. $(E_1 \ltimes E_2) \cup (E_1 \ltimes E_3) = E_1 \ltimes (E_2 \cup E_3)$

Example 2. Referring to our example database in Fig. 2, all the fragments of table P are empty except for $P_2 = \{t_6\}$ and $P_8 = \{t_5, t_7\}$; i.e., $P = P_2 \cup P_8$. Consider the set $F = \{P_7, P_8\}$. Observe that it is not possible to rewrite $Q_{P_7} \cup Q_{P_8}$ into a single FPJ$^+$ query using the above rules. However, since both

[3] Note that this example is not related to the example in Fig. 2.

fragments P_5 and P_6 are empty, we have $Q_{P_7} \cup Q_{P_8} = \bigcup_{i=5}^{8} Q_{P_i}$ which can now be rewritten into the FPJ$^+$ query $P \triangleright L$. Thus, $P \ltimes L \in Query(F)$. In addition, it is now possible to simplify Q_{P_8} in six ways. First, since $P_5 = \underline{P} \triangleright L \triangleright PS = \underline{P} \triangleright L \triangleright (PS \ltimes L) \triangleright (PS \triangleright L)$ is empty, $Q_{P_8} = Q_{P_8} \cup Q_{P_5}$ can be simplified using rule S1 to $SQ_8 = \underline{P} \triangleright L \triangleright (PS \ltimes L)$. Second, since P_4 is empty, $Q_{P_8} = Q_{P_8} \cup Q_{P_4}$ can be first rewritten using rule S2 to $(\underline{P} \ltimes L \cup \underline{P} \triangleright L) \triangleright (PS \ltimes L) \ltimes (PS \triangleright L)$ and further simplified using rule S1 to $SQ_7 = \underline{P} \triangleright (PS \ltimes L) \ltimes (PS \triangleright L)$.

Similarly, Q_{P_8} can be simplified to four other queries SQ_9, SQ_{11}, SQ_{10}, and SQ_{12}; the definitions of these queries are shown in Fig. 2i. The queries in $Query(\{P_2\})$ and $Query(\{P_2, P_8\})$ are shown in Figs. 2g and 2h, respectively. □

As the queries in $Query(F)$ are derived from the fragment queries, each query in $Query(F)$ also inherits the maximum query height of α from the fragment queries. Since there could be many queries in $Query(F)$, `QREF` uses a concise representation of $Query(F)$ to optimize both its storage as well as the efficiency of enumerating candidate projection table queries from $Query(F)$. Specifically, instead of storing $Query(F)$, `QREF` stores only the subset consisting the minimal and maximal queries (corresponding to the simplest and most complex queries), denoted by $ExtQuery(F)$ (for the extremal queries in $Query(F)$). The details of this optimization are given elsewhere [10] due to space constraint. Thus, the table fragment index for R_i, $TIndex(R_i)$, is a set of $(F, ExtQuery(F))$ entries where $F \subseteq Frag(R_i)$, $F \neq \emptyset$, and $ExtQuery(F) \neq \emptyset$. Fig. 2j shows the table fragment index for relation P in the running example. Note that in the implementation of $TIndex(R_i)$, each F is represented by a set of fragment identifiers.

5 Reverse Engineering PJ$^+$ Queries

In this section, we present the online phase of `QREF` which given an output table T consisting of k columns $C_1, \cdots, C_k$ and a database D, generates a target PJ$^+$ query Q such that $Q(D) = T$. `QREF` optimizes this QRE problem by using the table fragments, fragment query index, and table fragment index computed in the offline phase in three main steps.

First, given D and T, `QREF` enumerates a set of candidate projection tables for computing T from D. Second, for each set of candidate projection tables $PT = \{P_1, \cdots, P_k\}$, $k \geq 1$, `QREF` enumerates a set of candidate projection queries $PQ = \{Q_1, \cdots, Q_k\}$, where each Q_i is a FPJ$^+$ query w.r.t. P_i. Third, given a set of candidate projection queries PQ, `QREF` enumerates candidate queries by merging the queries in PQ. For each enumerated candidate query Q, `QREF` performs a query validation to check whether Q could indeed compute T from D. The algorithm terminates when either `QREF` finds a target query to compute T or all the enumerated candidate queries fail the validation check.

5.1 Generating Candidate Projection Tables

The first step in `QREF` is to generate a set candidate projection tables PT that could compute the given output table $T(C_1, \cdots, C_k)$. This is a common task

shared by all existing approaches that starts by identifying the set of covering attributes for each column C_i in the output table T. Based on the identified covering attributes for T, the next step is to enumerate all the sets of candidate projection tables. Since simpler target queries are preferred over more complex ones (in terms of query size), QREF enumerates the candidate projection table sets in non-descending order of the number of projection tables; for example, in Fig. 2l, the candidate set PT_1 will be considered before PT_2 and PT_3.

For each candidate projection table being enumerated; e.g., *P(pkey,brand)* to cover $T(C_1, C_2)$, it is necessary to validate that indeed $\pi_{pkey,brand}(P) \supseteq \pi_{C_1,C_2}(T)$. QREF optimizes this validation step by first performing a partial validation using a small random sample of tuples from T to check whether these tuples could be projected from the candidate projection table. If the partial validation fails, QREF can safely eliminate this candidate from further consideration.

5.2 Generating Candidate Queries

The second step in QREF is to generate candidate queries for each set $PT = \{R_1, \cdots, R_m\}$ of candidate projection tables enumerated by the first step. Recall that each R_i covers some subset of columns in T (i.e., $R_i(A_i)$ covers $T(A'_i)$ for some attribute lists A_i and A'_i) such that every column in T is covered.

Our query-centric approach performs this task using two main steps. First, QREF enumerates a set PQ_i of candidate projection table queries for each candidate projection table R_i in PT. Second, for each combination of candidate projection table queries $CPQ = \{Q_1, \cdots, Q_m\}$, where each $Q_i \in PQ_i$, QREF enumerates candidate queries that can be generated from CPQ using different ways to merge the queries in CPQ.

Generating Projection Table Queries. To generate candidate projection table queries for a projection table $R_i \in PT$, QREF uses the table fragment index $TIndex(R_i)$. Specifically, for each $(F, ExtQuery(F))$ entry in $TIndex(R_i)$, where $F \subseteq Frag(R_i)$, QREF determines whether $F(A_i)$ covers $T(A'_i)$. If so, each query in $ExtQuery(F)$ is a candidate projection table query for R_i.

Example 3. Consider the set of candidate projection tables $PT_1 = \{P(pkey, brand), S(name)\}$, where $P(pkey, brand)$ covers $T(C_1, C_2)$ and $S(name)$ covers $T(C_3)$. Among the three index entries in $TIndex(P)$ (shown in Fig. 2j), $F(pkey, brand)$ covers $T(C_1, C_2)$ when $F = \{P_8\}$. Therefore, queries in $ExtQuery(\{P_8\})$ are candidate projection table queries for P. □

Merging Projection Table Queries. Consider a set of candidate projection tables $PT = \{R_1, \cdots, R_m\}$ to compute T, where QREF has enumerated a projection table query Q_i for each R_i. We now outline how QREF enumerates different candidate queries from $\{Q_1, \cdots, Q_m\}$ by different ways to merge these queries. To merge the queries $\{Q_1, \cdots, Q_m\}$ into a candidate query Q, QREF first identifies a query node v_i in each query Q_i such that all the v_i's can all be merged into a single node v to connect all the Q_i's into a single query Q. We refer to each of the v_i's as a **mergeable node**, and to the combined query Q as a **merged query**. The details of the query merging processing are described elsewhere [10].

Example 4. Consider the merging of two projection table queries: $Q_1 = \underline{P1} \ltimes PS1 \triangleright L1$ and $Q_2 = \underline{S1} \ltimes PS2 \triangleright L2$, whose query graphs are shown in Fig. 2m and Fig. 2n. There is only one choice of mergeable nodes, i.e. $\{PS1, PS2\}$. The merged query generated by merging $\{PS1, PS2\}$ is shown in Fig. 2o, where the merged node is denoted by PS and all the semijoin edges along the path connecting the projection nodes $S1$ and $P1$ are converted to inner join edges. □

Generating Candidate Queries. Each merged query Q generated by `QREF` is a candidate query, and Q itself could be used to derive possibly additional candidate queries by combining different subsets of mergeable nodes in Q. Thus, for a given set of candidate projection tables $PT = (R_1, \cdots, R_m)$ to compute T, `QREF` enumerates different candidate queries w.r.t. PT by applying different levels of enumeration. First, `QREF` enumerates different projection table queries for each R_i. Second, for each combination of projection table queries $PQ = \{Q_1, \cdots, Q_m\}$, `QREF` enumerates multiple merged queries from PQ via different choices of mergeable nodes. Third, for each merged query Q, `QREF` derives additional candidate queries from Q by enumerating different ways to further merge the query nodes in Q. As simpler target queries (with fewer tables) are preferred over more complex target queries, `QREF` uses various heuristics to control the enumeration order to try to generate simpler candidate queries earlier. For example, for enumerating combinations of projection table queries $PQ = \{Q_1, \cdots, Q_m\}$, `QREF` uses the sum of the query size (i.e., $\sum_{i=1}^{m} |Q_i|$) as a heuristic metric to consider those queries with smaller size earlier.

5.3 Validation of Candidate Queries

For each candidate query Q enumerated, `QREF` needs to validate whether Q could indeed compute T. In this section, we explain how `QREF` optimizes this validation process with the help of the table fragments and the fragment query indexes. The query validation process consists of two steps. First, `QREF` performs a less costly *partial validation* of Q by randomly choosing a small sample of tuples from the output table T and checking if these tuples are contained in Q's result. If the outcome is conclusive (i.e., one of the chosen tuples can not be produced by Q), then `QREF` concludes that Q is not a target query and the validation process terminates. On the hand, if the partial validation is not conclusive, `QREF` will resort to a more costly *full validation* to compute the result of Q on the database and check whether the result is equal to T.

`QREF` is able to optimize the evaluation of Q using table fragments for two reasons. First, some of joins with filter nodes in Q can be eliminated as they are already precomputed in the table fragments. As an example, consider the query Q_1 shown in Fig. 1b which involves a 5-table join: two projection nodes (S1 and S2), one connection node (N), and two filter nodes (L1 and L2). For this query, `QREF` is able to eliminate the join $\underline{S1} \ltimes L1$ by accessing the fragments of table S that satisfy FPJ$^+$ query expression $\underline{S} \ltimes L$ (instead of the entire table S). Similarly, `QREF` is able to eliminate the join $\underline{S2} \triangleright L2$ by accessing the fragments of table S that satisfy the FPJ$^+$ query expression $\underline{S} \triangleright L$.

Second, for joins that cannot be eliminated (i.e., joins with projection/connection nodes), QREF can optimize the evaluation by accessing only relevant table fragments instead of entire tables. Referring again to the query Q_1 in Fig. 1b, consider the table N. Instead of accessing the entire table N, QREF only need to access the fragments of table N that match the FPJ$^+$ query $Q' = \underline{N} \ltimes (S1 \ltimes L1) \ltimes (S2 \triangleright L2)$ assuming $\alpha = 2$, and the matching fragments for Q' can be efficiently identified using the fragment index $QIndex(N)$. Specifically, QREF searches the trie index $QIndex(N)$ twice using the two root-to-leaf paths in Q' ($\underline{N} \ltimes (S \ltimes L1)$ and $\underline{N} \ltimes (S \triangleright L2)$)). By intersecting the two lists of fragment identifiers associated with the paths' leaf nodes, QREF can efficiently identify the set of matching table fragments in N. Depending on the selectivity of Q', this alternative evaluation plan that retrieves only relevant table fragments could be more efficient than a full table scan.

6 Experiments

In this section, we present the results of a performance evaluation of our proposed approach, QREF. Our study compares QREF against our main competitor STAR [20] as STAR is designed for PJ queries, which is the closest query class to the PJ$^+$ queries supported by QREF[4]. Our results show that QREF generally outperforms STAR by an average improvement factor of 120 and up to a factor of 682.

We conducted four experiments using the same TPC-H benchmark database as [20], whose database size is 140MB generated using Microsoft's skewed data generator for TPC-H. All the experiments were performed on an Intel Xeon E5-2620 v3 2.4GHz server running Centos Linux 3.10.0 with 64GB RAM, and the database was stored on a separate 1TB disk using PostgreSQL 10.5.

Experiments 1 and 2 are based on the same set of queries used by STAR [20], where Experiment 1 consists of 22 PJ queries (TQ1 to TQ22) modified from the 22 TPC-H benchmark queries, and Experiment 2 consists of 6 queries (Q1 to Q6) created by [20]. Experiment 3 consists of 5 queries (QQ1 to QQ5) shown in Fig. 4a, which were created by us to showcase the strengths of QREF compared to STAR. Experiment 4 is designed to evaluate the effectiveness of QREF's fragment-based query validation; the results are given in [10] due to space constraint.

For each query Q, we first execute Q on the database to compute its result T and then measure the time taken to reverse engineer Q from T using each of the approaches. To be consistent with the experimental timings reported in [20], the execution timings for all approaches exclude the following components: preprocessing time and the validation of the last candidate query. Each of the timings (in seconds) reported in our experiments is an average of five measurements. If an approach did not complete after running for one hour, we terminate

[4] We did not compare against FastQRE [8] for two reasons. First, FastQRE supports only CPJ queries which are even more restrictive than PJ queries. Second, the code for FastQRE is not available, and its non-trivial implementation requires modification to a database system engine to utilize its query optimizer's cost model for ranking candidate queries.

QID	Query	Projection
QQ1	$S \bowtie (N \triangleright C)$	S.name, N.name
QQ2	$C \ltimes (N \ltimes S) \triangleright O$	C.name
QQ3	$(L1 \rtimes S1) \bowtie N \bowtie (S2 \ltimes L2)$	S1.name, S2.name
QQ4	$(C \rtimes N) \bowtie (S \ltimes L)$	N.name, S.name
QQ5	$N \bowtie C \bowtie O \bowtie L$	C.name O.orderdate L.shipdate

(a) Expt. 3: Queries

QID	1	2	3	4	5	6	7	8	9	10	11
STAR	8.2	E	38.6	22.42	E	8.42	E	E	E	E	34.5
QREF	0.28	0.5	1.78	0.45	16.32	0.2	40.1	24.6	0.4	0.97	2.89
f	29.3	-	21.7	49.8	-	42.1	-	-	-	-	11.9
QID	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
STAR	27.27	2.2	16.6	82.21	80.09	48.4	14.0	E	E	E	4.87
QREF	0.04	2	0.38	0.23	0.5	0.41	2.11	0.57	0.8	294	0.13
f	681.8	1.1	43.7	357.4	160.2	118	6.6	-	-	-	37.5

(b) Expt. 1: Timing Results

QID	1	2	3	4	5	6	7	8	9	10	11
STAR	1	-	20	4	-	1	-	-	-	-	5
QREF	1	1	1	1	3	1	101	92	1	1	1
QID	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
STAR	6	1	11	225	11	8	9	-	-	-	6
QREF	1	1	1	1	1	1	1	1	1	263	1

(c) Expt. 1: # of Candidate Queries

QID	Q1	Q2	Q3	Q4	Q5	Q6
STAR.time	21.86	56.76	T	51.91	180.22	311.5
QREF.time	0.54	1.15	3.03	1.47	16.49	10.91
f	40.48	49.36	-	35.31	10.93	28.55
STAR.#cand	4	4	3	17	15	15
QREF.#cand	1	1	2	9	5	6

(d) Expt. 2: Timing Results

QID	QQ1	QQ2	QQ3	QQ4	QQ5
STAR	E	E	E	24.74	20.9
QREF	0.018	0.11	4.12	0.64	0.052
f	-	-	-	38.7	401.9

(e) Expt. 3: Timing Results

number	1	2	3	4
STAR.time	25.36	127.7	E	E
QREF.time	0.05	0.13	0.2	0.2
f	507.2	982.3	-	-
STAR.#cand	40	250	-	-
QREF.#cand	1	1	1	1

(f) Expt. 3: Effect of # Covering attributes

Fig. 4: Experimental Results (timings in seconds)

its execution and indicate its timing value by "E". For each of our experimental queries, both STAR and QREF are able to find the same target query if they both complete execution within an hour.

Both QREF and STAR were implemented in C++[5]. We run approach STAR with its default settings and turn on all its optimization techniques. For QREF, we set $\alpha = 2$ and always use table fragments for query validation. QREF took slightly under a minute to partition each database table into fragments, and build all the table fragment and fragment query indexes. The number of non-empty fragments created by QREF for tables Orders, Nation, Part, Region, Supplier, LineItem, PartSupp, and Customer are 2, 9, 3, 4, 3, 1, 5, and 6, respectively.

Experiment 1. This experiment compares the performance of QREF and STAR to reverse engineer the 22 TPC-H queries. The speedup factor f is defined to be the ratio of STAR's timing to QREF's timing. Fig. 4b compares the execution time. STAR was unable to find queries TQ2, TQ5, TQ6, TQ7, TQ8, TQ9, TQ10, TQ19, TQ20, and TQ21 within one hour. The reason is that STAR is very memory intensive, and its executions ran out of memory for these queries. This behaviour of STAR was also observed by [8]. Specifically, TQ5, TQ7, TQ8, and TQ21 are large queries involving between 6 to 8 tables. STAR did not perform well for these queries as it spent a lot of time generating and validating complex candidate

[5] The code for STAR was based on a version obtained from the authors of [20].

queries. In contrast, QREF could find all these complex queries even though it took longer compared to the timings for the other simpler queries.

For queries TQ2, TQ9, TQ10, TQ19, and TQ20, QREF outperformed STAR significantly for these queries. One common property among these queries is that they all have many output columns: the number of output columns for queries TQ2, TQ9, TQ10, TQ19 and TQ20 are 12, 7, 11, 8, and 7, respectively. For query TQ2, it even has 5 output columns that are projected from the same Supplier table. For the remaining queries where both STAR and QREF can find the target queries, QREF always outperformed STAR; the minimum, average, and maximum speedup factors of QREF over STAR for these queries are 1.12, 120, and 682.1, respectively. Fig. 4c compares the number of candidate queries generated by STAR and QREF for the 22 queries in Experiment 1. The results clearly demonstrate the effectiveness of QREF's query-centric approach to enumerate candidate queries compared to STAR's path-centric approach.

Experiment 2. This experiment (shown in Fig. 4d) compares the performance using queries Q1-Q6 in [20]. Observe that QREF outperforms STAR by a minimum, average, and maximum speedup factor of 11, 33, and 49, respectively. Similar to Experiment 1, QREF also generated fewer candidate queries compared to STAR.

Experiment 3. This experiment (shown in Fig. 4e) compares the performance using the five queries in Fig. 4a. Both queries QQ1 and QQ2 consist of an antijoin, and STAR was unable to reverse engineer such queries as expected; in contrast, QREF was able to find these target queries very quickly. Queries QQ3 and QQ4 both consist of filtering nodes, and the results show that QREF was able to significantly outperform STAR for target queries containing filtering nodes. In particular, for query QQ3, STAR's path-centric approach actually requires generating join paths of up to length 3 for it to find the target query; however, this entails enumerating a large search space of join paths and the execution of STAR was terminated after running for one hour and generating 232 candidate queries without finding the target query.

Finally, Fig. 4f examines the effect of the number of covering attributes in a table using four variants of query QQ5. The first variant has only one column C.name from the Customer table, and the additional variants are derived by progressively adding more projection columns (C.acctbal, C.address, and C.phone) from Customer table. Thus, the i^{th} variant projects i columns from Customer table. Observe that as the number of covering attributes increases, both the execution time and the number of generated candidate queries increased significantly for STAR; indeed, when the number increases to more than 2, STAR had to enumerate hundreds of candidate queries and its executions did not terminate after running for more than an hour. In contrast, the performance of QREF is less negatively affected by the the number of covering attributes.

7 Conclusions

In this paper, we focus on the QRE problem for PJ^+ queries, which is a more expressive class of queries than PJ queries by additionally considering antijoins

as well as inner joins (and semijoins). To enhance efficiency, we propose a novel query-centric approach that is based on horizontally partitioning each database into fragments using FPJ^+ queries. By associating each table fragment with a FPJ^+ fragment query, our approach is amenable to using efficient precomputation and index techniques to both generate candidate projection table queries as well as validate candidate queries. Our experimental study demonstrates that our approach significantly outperforms the state-of-the-art solution (for PJ queries without antijoins) by an average improvement factor of 120.

Acknowledgements We would like to thank Meihui Zhang for sharing the code of `STAR`. This research is supported in part by MOE Grant R-252-000-A53-114.

References

1. Abouzied, A., Angluin, D., Papadimitriou, C., Hellerstein, J.M., Silberschatz, A.: Learning and verifying quantified boolean queries by example. In: PODS (2013)
2. Arenas, M., Diaz, G.I.: The exact complexity of the first-order logic definability problem. ACM TODS **41**(2), 13:1–13:14 (May 2016)
3. Bonifati, A., Ciucanu, R., Staworko, S.: Learning join queries from user examples. ACM TODS (2016)
4. Das Sarma, A., Parameswaran, A., Garcia-Molina, H., Widom, J.: Synthesizing view definitions from data. In: ICDT (2010)
5. Gao, Y., Liu, Q., Chen, G., Zheng, B., Zhou, L.: Answering why-not questions on reverse top-k queries. PVLDB (2015)
6. He, Z., Lo, E.: Answering why-not questions on top-k queries. In: ICDE (2012)
7. He, Z., Lo, E.: Answering why-not questions on top-k queries. TKDE (2014)
8. Kalashnikov, D.V., Lakshmanan, L.V., Srivastava, D.: Fastqre: Fast query reverse engineering. In: SIGMOD (2018)
9. Li, H., Chan, C.Y., Maier, D.: Query from examples: An iterative, data-driven approach to query construction. PVLDB (2015)
10. Li, M., Chan, C.Y.: Efficient query reverse engineering using table fragments. Tech. rep. (2019)
11. Liu, Q., Gao, Y., Chen, G., Zheng, B., Zhou, L.: Answering why-not and why questions on reverse top-k queries. VLDB Journal (2016)
12. Luo, Y., Fletcher, G.H.L., Hidders, J., Wu, Y., Bra, P.D.: External memory k-bisimulation reduction of big graphs. In: ACM CIKM. pp. 919–928 (2013)
13. Panev, K., Michel, S., Milchevski, E., Pal, K.: Exploring databases via reverse engineering ranking queries with paleo. PVLDB (2016)
14. Psallidas, F., Ding, B., Chakrabarti, K., Chaudhuri, S.: S4: Top-k spreadsheet-style search for query discovery. In: SIGMOD (2015)
15. Shen, Y., Chakrabarti, K., Chaudhuri, S., Ding, B., Novik, L.: Discovering queries based on example tuples. In: SIGMOD (2014)
16. Tan, W.C., Zhang, M., Elmeleegy, H., Srivastava, D.: Reverse engineering aggregation queries. PVLDB (2017)
17. Tran, Q.T., Chan, C.Y.: How to conquer why-not questions. In: SIGMOD (2010)
18. Tran, Q.T., Chan, C.Y., Parthasarathy, S.: Query by output. In: SIGMOD (2009)
19. Weiss, Y.Y., Cohen, S.: Reverse engineering spj-queries from examples. In: PODS (2017)
20. Zhang, M., Elmeleegy, H., Procopiuc, C.M., Srivastava, D.: Reverse engineering complex join queries. In: SIGMOD (2013)