FastQRE: Fast Query Reverse Engineering

Dmitri V. Kalashnikov
AT&T Labs Research
dvk@research.att.com

Laks V.S. Lakshmanan*
University of British Columbia
laks@cs.ubc.ca

Divesh Srivastava
AT&T Labs Research
divesh@research.att.com

ABSTRACT

We study the problem of *Query Reverse Engineering* (QRE), where given a database and an output table, the task is to find a simple project-join SQL query that generates that table when applied on the database. This problem is known for its efficiency challenge due to mainly two reasons. First, the problem has a very large search space and its various variants are known to be NP-hard. Second, executing even a single candidate SQL query can be very computationally expensive. In this work we propose a novel approach for solving the QRE problem efficiently. Our solution outperforms the existing state of the art by 2–3 orders of magnitude for complex queries, resolving those queries in seconds rather than days, thus making our approach more practical in real-life settings.

CCS CONCEPTS

• **Theory of computation → Data integration**;

KEYWORDS

Automated Data Lineage Discovery, Column Coherence, CGM

ACM Reference Format:
Dmitri V. Kalashnikov, Laks V.S. Lakshmanan, and Divesh Srivastava. 2018. FastQRE: Fast Query Reverse Engineering. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3183713.3183727

1 INTRODUCTION

Query Reverse Engineering (QRE) is a well-studied problem which arises frequently in practice [5, 6, 8, 18, 22, 27, 30]. Given table R_{out} and dataset $\mathcal{D}$ the task is to find a *generating* query Q_{gen} that when applied on $\mathcal{D}$ generates R_{out}. In this paper we focus on simple project-join (PJ) `SQL` queries and propose a highly efficient approach for reverse engineering of such queries.

QRE problem arises, for example, when a business/data analyst finds a useful table R_{out} which she wants to augment. Table R_{out} can be a business report stored as an excel or doc file, or as a table in a database. The analyst knows that R_{out} has been generated by some query Q_{gen} on database $\mathcal{D}$, and wants to find Q_{gen} and change it according to her needs. However, it is not uncommon that the generating query Q_{gen} is no longer known: *e.g.,* the person who has created Q_{gen} cannot be identified, or is not available, etc. Thus, the query needs to be reverse engineered.

In general, techniques developed to solve the various variants of the QRE problem can also be leveraged to solve other data management problems. For example, in a *data integration* task the analyst might want to specify the schema of a table she wants to create as well as a few sample tuples this table should contain. QRE approach then would find a query that, when applied on the database, would generate the desired table containing the sample tuples. QRE solution can facilitate the tasks of *data lineage discovery*, database usability, data exploration, and data analysis [8, 27, 30, 31].

Two most studied versions of QRE are its *exact* and *superset* variants, which look for Q_{gen} such that $Q_{gen}(\mathcal{D}) = R_{out}$ and $Q_{gen}(\mathcal{D}) \supseteq R_{out}$, respectively. The superset QRE is known to be simpler, as it is sufficient to consider queries whose graphs form trees instead of generic graphs. In terms of the space of solutions available for these two variants, we have:

(1) *Solutions for Exact QRE.* First, we have solutions for the exact QRE, like [38]. Often, they can be easily modified to solve superset QRE as well. The approach in [38] is a good overall solution that methodically goes over the entire search space. However, it can be slow (*e.g.,* take days to do QRE) for complex queries and large databases. In some cases, not being able to resolve a query in a reasonable amount of time is equivalent to failing on resolving that query. In summary, the class of queries this approach can resolve is very broad, but the speed for some cases can be slow.

(2) *Solutions for Superset QRE.* Then, we have solutions that solve the superset variant, like [27]. For them we can observe the reverse trend: they can be fast but the class of queries they can discover and the application domain are very narrow compared to the above. These techniques are not designed to handle exact QRE and, in general, cannot be easily modified to solve it.

(3) `FastQRE` *Approach.* Our `FastQRE` approach occupies a unique niche: it is closer to (1) in terms of the class of queries it can handle, but closer to (2) in terms of speed. That is, the approach can handle both the exact and superset QRE variants. The class of queries it can handle is quite broad, as discussed in Section 3, yet the approach can be orders of magnitude faster than (1) on complex queries and large databases.

The efficiency challenge of QRE mainly comes from two factors: the cost of querying over large tables and its large search space. The first factor distinguishes QRE the most from many problems studied in the related work: any QRE technique should scale to large tables. Running a large number of checks on whether $Q(\mathcal{D}) = R_{out}$ or $Q(\mathcal{D}) \supseteq R_{out}$ for each candidate query Q is simply unacceptable. Thus, techniques that operate by enumerating over many candidate queries, frequently studied in the literature for related problems, simply would not work well for QRE. Instead, smart techniques should be designed to avoid as many of such checks as possible.

*Work done while visiting AT&T Labs Research.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3183727

Another challenge of solving QRE is its large search space, as its many variants are known to be NP-hard [38]. The problem is challenging due to two types of ambiguity that need to be resolved: *column* and *join-path* ambiguity. Column ambiguity arises because for each column in R_{out} we need to find the column and the table instance in database $\mathcal{D}$ that it has been generated from, which are called *projection* column and projection table instance respectively. Join-path ambiguity arises because we need to provide a way to interconnect the discovered projection table instances via the correct join paths. Addressing these two types of ambiguities could take days for some state of the art techniques, even for smaller databases.

To deal with the efficiency challenge we develop a new `FastQRE` framework, whose overall architecture is illustrated in Figure 6. The framework consists of four modules that deal with preprocessing, candidate query generation, query validation, and feedback, where each module in turn has several subcomponents. While the purpose of these modules and their components will be described in detail in Section 4, we highlight two of the components below.

First, the *Direct Column Coherence* component leverages the concept of *direct* column coherence to address column ambiguity (Section 4.2). If a group of columns from R_{out} has been generated by Q_{gen} from a group of columns from some table R, then the tuples in these columns of R_{out} should be a subset of tuples over the columns from R. We call this property *(direct) column coherence*. Our solution is based on a related insight that if a group of columns in some table R is coherent, then there is a good chance that the corresponding columns in R_{out} have been generated from that group. Our proposed solution actively uses this intuition for resolving column ambiguity (Section 2). It first discovers coherent column groups and stores them as tuples called CGMs, which stands for coherent, column group, and mapping. The *Ranking Column Mapping* component then uses these CGMs to rank different possible combinations (mappings) of what the correct projection columns could be. The ranking function developed by us proves to be highly effective for the task.

Second, the *Indirect Column Coherence* component employs *(indirect) column coherence* checks to reduce the join-path ambiguity (Section 4.5). Recall that the task of the overall algorithm is to find a generating query Q_{gen}. The algorithm forms Q_{gen} by discovering certain join paths and then merging them together. Each join path itself corresponds to a subquery that if executed would result in some relation R. In Section 4.5 we will see that for a join path to be a part of Q_{gen}, the columns in this R must be coherent with respect to some columns in R_{out}. If they are not coherent, the join path can be safely filtered away from further consideration.

Overall, the **novel contributions** of this paper are:

- Notion of direct column coherence and CGMs (Section 4.2).
- Ranking mechanism that leverages CGMs for addressing column ambiguity (Section 4.3).
- Ranking mechanism of composing candidate queries (Section 4.4)
- Notion of indirect column coherence for addressing join-path ambiguity (Section 4.5).
- An extensive empirical evaluation of our solution (Section 5).

Before describing our main approach in Section 4, we first present a motivating example in Section 2, and then introduce the notation and formally define the problem in Section 3.

2 MOTIVATING EXAMPLES

To motivate the problem and our solution we will use two examples. The first example is generic and will help us to illustrate most of the concepts in the paper as well as to demonstrate the various complexities that can arise in practice. The second example is more specific and we use it to introduce the notion of column coherence.

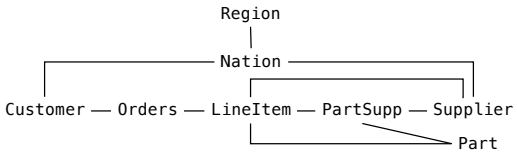


Figure 1: Schema graph for TPC-H.

Example 2.1 (Running example). Let us consider the well-studied TPC-H benchmark dataset [29]. Its schema graph is illustrated in Figure 1, where the nodes represent tables in the schema and the edges represent the possible joins between tables. TPC-H has eight tables: LineItem, Orders, Customer, Nation, Region, PartSupp, Supplier, and Part, commonly abbreviated as: L, O, C, N, R, PS, S, and P, respectively.

```
SELECT S1.suppkey, S1.name, PS1.availqty, S2.suppkey, S2.name
FROM   Supplier S1, Supplier S2, Partsupp PS1, Partsupp PS2, Part P, Nation N
WHERE  S1.suppkey=PS1.suppkey AND S2.suppkey=PS2.suppkey AND
       P.partkey=PS1.partkey AND P.partkey=PS2.partkey AND
       N.nationkey=S1.nationkey AND N.nationkey=S2.nationkey
```

Figure 2: SQL of Query 1. Query 2 is the same but without `PS1.availqty` attribute in its `SELECT` clause.

We will consider two closely related queries: Queries 1 and 2, see Figure 2. Query 2 finds all pairs of suppliers located in the same nation and supplying the same part. Query 1 is like Query 2, except it also reports the available quantity of each such common part for the first supplier in the pair.

Each of these queries contains two instances of tables S and PS: S1, S2, PS1, and PS2. The `SELECT` clause of Query 1 lists five columns, called *projection columns*, see Figure 2. Correspondingly, these columns belong to two *projection tables* S and PS and three *projection table instances* S1, S2, and PS1. Query 2 is similar, but does not have `PS1.availqty` in its `SELECT` clause.

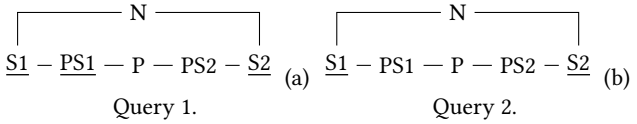


Figure 3: Query graphs for Queries 1 and 2.

The query graphs for Queries 1 and 2 are shown in Figure 3. The nodes in the graph correspond to the instances of tables used in the query, where the projection table instances are underlined. The edges correspond to the joins used by the query.

A	B	C	D	E
1	Supplier#000000001	380	264	Supplier#000000264
1	Supplier#000000001	976	270	Supplier#000000270
5	Supplier#000000005	4919	471	Supplier#000000471
8	Supplier#000000008	7085	269	Supplier#000000269
15	Supplier#000000015	1596	748	Supplier#000000748
⋮	⋮	⋮	⋮	⋮

Table 1: A spreadsheet with R_{out} table for Query 1.

Assume an excel spreadsheet containing R_{out} table shown in Table 1, which has been generated by Query 1, whereas R_{out} for Query 2 is not shown. Given the TPC-H dataset $\mathcal{D}$ and R_{out}, our QRE task is to find the generating query Q_{gen} that when applied on $\mathcal{D}$ generates R_{out}. □

The next example motivates the concept of column coherence.

A	B	C
1	2	2
2	4	3
3	2	1

(a) Table $R1$

D	E
1	a7
2	a2
3	a1

(b) Table $R2$

F	G
2	b3
3	b5

(c) Table $R3$

X	Y	Z	W
1	2	a1	b5
3	4	a2	b3

(d) Table R_{out}

C	B
2	2
3	4
1	2

(e) $\pi_{C,B}(R1)$

X	Y
1	2
3	4

(f) $\pi_{X,Y}(R_{out})$

Figure 4: Column Coherence.

Example 2.2 (Column Coherence). Figure 4 shows a toy database $\mathcal{D}_{toy}$ that consists of three tables $R1$, $R2$, and $R3$. Column A is the primary key for $R1$ and columns D in $R2$ and F in $R3$ are the corresponding foreign keys that point to A. Table R_{out} has been generated by query Q_{gen}, which is:

```
SELECT C as X, B as Y, E as Z, G as W
FROM   R1, R2, R3
WHERE  R2.D = R1.A AND R3.F = R1.A
```

Given R_{out} and $\mathcal{D}_{toy}$, our goal is to reverse engineer Q_{gen}.

Preprocessing. Initially, without any prior analysis, we should assume that columns X, Y, Z, and W from R_{out} could have been generated from any of the columns $\{A, B, C, D, E, F, G\}$ in $\mathcal{D}$, which creates too many combinations. The actual names of columns of R_{out}, when present, might help reduce this ambiguity. However, the names might not match, or might be absent, or ambiguous, or too generic. Thus, it is desirable to reduce the ambiguity associated with each column in an automated fashion.

For that goal, we can use the standard technique of computing the column cover. Let the notation $R.a \mapsto R_{out}.c$ denote the fact that column $R_{out}.c$ could have been generated from column $R.a$. Now let us observe that column X contains value "1" which is not in column B. Thus, X could not have been generated from B by a PJ SQL query. More generally, we can state that $B \not\mapsto X$ because $\pi_X(R_{out}) \nsubseteq \pi_B(R1)$.

Using such a set containment property, we can compute for each column $c \in R_{out}$ its *column cover* $S_c = \{R.a : \pi_a(R) \supseteq \pi_c(R_{out})\}$, which is the set of all columns whose values are superset of values of c. It represents all columns that column c could have been generated from with respect to the set containment property. In our example, using this method we can compute $S_X = \{A, C, D\}$, $S_Y = \{B\}$, $S_Z = \{E\}$, and $S_W = \{G\}$.

Column Coherence and CGMs. Notice how the above step does not resolve column ambiguity fully, even for this toy dataset. First, we still need to choose the correct projection column for X from S_X out of 3 remaining combinations. Second, assume we even somehow know that $S_X = \{C\}$. Then, since columns B and C are both from table $R1$, we still will need to decide if B and C come from the same *instance* of $R1$, or from two distinct instances. We will use notation $R1(B, C)$, or just (B, C), when we want to emphasize B and C come from the same *instance* of $R1$.

Analyzing column coherence can help in addressing the aforementioned ambiguity. We will essentially extend the single-column logic used in the preprocessing step to multiple columns. We have $S_X = \{A, C, D\}$ and assume we want to check if columns (X, Y) could have been generated from columns $R1(A, B)$. We can check that $\pi_{X,Y}(R_{out}) \nsubseteq \pi_{A,B}(R1)$: *e.g.,* while tuple (1,2) from (X, Y) columns in present in $R1(A, B)$, tuple (3,4) is not present there. Thus $R1(A, B) \not\mapsto (X, Y)$, because tuple (3,4) cannot be generated this way. However, for the pair $R1(C, B)$ it holds that $\pi_{X,Y}(R_{out}) \subseteq \pi_{C,B}R1$, that is, columns C and B are *coherent* with respect to (X, Y). Among all column pairs, C and B is the only coherent pair. Notice how it coincides with the fact that Q_{gen} uses $R1(C, B)$ as the projection columns to get (X, Y) in R_{out}!

Our solution will leverage the insight that if a group of columns is coherent, then it is likely that it is not by chance, especially for large tables with diverse set of values, and large column groups. For example, this intuition tells us that, among many possible *column mappings* for columns of R_{out}, we should perhaps try first the mapping where $(X, Y) \leftrightarrow R1(C, B)$, $Z \leftrightarrow R2(E)$ and $W \leftrightarrow R3(G)$. The algorithm thus finds coherent column groups, such as $R1(C, B)$, $R2(E)$, and $R3(G)$, and stores them as tuples called CGMs, which are then used to rank candidate column mappings.

Indirect Column Coherence. Interestingly, the above logic can be extended even further to handle join-path ambiguity. We can see that $R1, R2$ and $R3$ are all projection tables. We need to decide how to interconnect them to form Q_{gen}. Given the above discussion, it is reasonable to check if $R1(C, B)$ and $R2(E)$ are involved in Q_{gen}. Since $R1$ and $R2$ can be joined directly via the primary-foreign key (pk-fk) $R1.A = R2.D$ condition, we can try that first, projecting the result on the attributes C, B, E. Let Q be that corresponding query and R be the resulting relation. Can query Q be a subpart of Q_{gen} query? Can $R1$ and $R2$ be connected via this direct join path in Q_{gen}?

Extending the previous logic, we can see that *if* $\pi_{X,Y,Z}(R_{out}) \nsubseteq R$ then Q cannot be part of Q_{gen}. However, in this specific case it holds that $\pi_{X,Y,Z}(R_{out}) \subseteq R$ and thus we cannot dismiss considering Q as subquery of Q_{gen}. This is indeed as expected because in our Q_{gen} this is how $R1$ and $R2$ are connected.

In general, this join path corresponds to a walk in the database schema graph, and such a check for *walk coherence* can filter away many wrong candidate queries. Once an incoherent walk is discovered, candidate queries that contain this walk will either be filtered away or not constructed in the first place. □

The above example demonstrates some basic intuition behind using the notion of column coherence. In the subsequent sections we will formally present our solution that leverages this basic intuition to address the QRE problem.

3 PRELIMINARIES

In this section, we introduce the notation and formally define the Query Reverse Engineering problem.

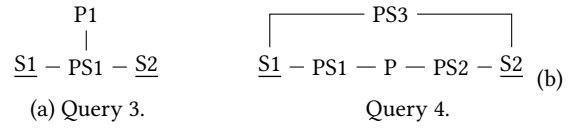
Schema Graph. Let $\mathcal{R} = \{R_1, R_2, \ldots, R_{|\mathcal{R}|}\}$ be the set of all relations/tables in database $\mathcal{D}$. Database $\mathcal{D}$ is commonly represented by its *schema graph* $G_S = (V_S, E_S)$, where V_S is a set of nodes and E_S is a set of edges. G_S is a labeled graph where each node in V_S corresponds to a distinct table R_i from $\mathcal{R}$. We will refer to the nodes by the corresponding table name R_i. A presence of an edge (R_i, R_j) in E_S indicates that a join is possible between tables R_i and R_j. For example, Figure 1 shows that $L \bowtie O$ is possible in a query, but $L \bowtie N$ is not possible. The label on the edge (used by our approach, but not shown in the figures for clarity) indicates which attributes/columns from R_i and R_j are involved in the join, as such a join in general might happen over different sets of columns. Thus G_S might contain parallel edges for multiple join keys as well as self-loops. We will refer to primary and foreign key by pk and fk. Our approach applies to any G_S irrespective of how its edges have been generated. As common, in our empirical study we will focus on the case where the edges correspond to all possible pk-fk joins.

Query. A project-join[1] (PJ) SQL query Q on $\mathcal{D}$ might involve multiple *instances* of the same table. For example, Query 1 involves two instances of Supplier (S) table: $S1$ and $S2$; as well as two instances of PartSupp (PS) table: $PS1$ and $PS2$. Let R_i^k denote the k-th instance of table R_i. If Q involves a single instance of R_i, for simplicity we will refer to it just as R_i, dropping $k = 1$.

If column $c \in R_{out}$ has been generated from column c_1 of table R_i, then c_1 is called the *projection column* for c and R_i is its *projection table*. We will use notation $c_\pi(c)$ and $R_\pi(c)$ to refer to the projection column and projection table of c. For our running example, `Supplier` table and its `name` column are examples of projection table and column. Similarly, the instance R_i^k of R_i from which column c has been generated is called *projection table instance* and denoted as $I_\pi(c)$.

For example, PS1 is a projection table instance, but PS2 is not. Notice, two columns of R_{out} that map into the same projection table R_i can either be from the same or two distinct instances of R_i. For example, columns A and B of R_{out} are generated from the same instance S1 of S, whereas columns A and D are generated from two distinct instances S1 and S2 of S. We will refer to non-projection tables (and table instances) also as *intermediate* tables (table instances). For Query 1, table PS is both projection and non-projection table, because PS1 is a projection and PS2 is a non-projection table instance.

[1]The `WHERE` clause of a PJ SQL query consists of only (pk-fk) join conditions, but no other selection conditions on attributes.



(a) Query 3. Query 4.

Figure 5: Queries 3 and 4.

Query Graph. Query Q is often represented by its *query graph* $G_Q = (V_Q, E_Q)$, where V_Q is the set of nodes and E_Q is the set of edges in G_Q. The graph is labeled and its nodes in V_Q correspond to *instances* of tables R_i^k involved in query Q. A presence of edge (R_i^k, R_j^ℓ) indicates that Q joins instances R_i^k and R_j^ℓ. For example, since edge $S1 - N$ is present in G_Q for Query 1, it means Query 1 includes a join of $S1$ and N. The label on an edge (not shown in our figures for clarity) indicates which columns are involved in the join – if the join could happen over different sets of columns. Naturally, edge (R_i^k, R_j^ℓ) cannot exist in G_Q if $(R_i, R_j) \notin G_S$. Also, if an edge is present in G_S it does not mean it will be present in G_Q. For example, for Query 1, edge $N - C$ is present in G_S but not in G_Q.

Nodes in V_Q are either *projection* or *intermediate* nodes based on whether they correspond to a projection table instances. For Query 1 the projection nodes are $S1$, $PS1$, $S2$. The rest are intermediate nodes.

CPJ query class. Let us define the class of *Covering PJ (CPJ)* queries as PJ queries satisfying the following two covering conditions defined on the query graph G_Q.

Consider all simple paths that exist between any pair of projection nodes in the query graph G_Q, but do not include other projection nodes. The first covering condition is that these paths should fully cover the entire graph G_Q. Query 3 in Figure 5 is an example of a query where this condition is violated. Its projection nodes are $S1$ and $S2$. The only simple path between them is $S1 - PS1 - S2$, which does not cover/include node $P1$ of the query graph.

The second covering condition is that if the *intermediate* nodes of G_Q contain at least two distinct instances of the same table, then all of these instances should be covered by (*i.e.,* be located/included on) a single path. Query 4 in Figure 5(b) is is an example of where the second condition does not hold, as PS1 and PS3 are located on two different paths.

The CPJ query class is very broad. For example, Queries 1 and 2 are CPJ queries, even though they (a) involve multiple instances of tables; and (b) their query graphs contain loops. The FastQRE approach can resolve any CPJ query.

Column Mapping. A *column mapping* $\mathcal{M}$ maps each column c from R_{out} into some $R_i^k.a$, that is, some column a of some table instance R_i^k. A column mapping that maps each c into $c_\pi(c)$ and $I_\pi(c)$ is called the *correct mapping*. For example, for Query 1 the correct mapping $\mathcal{M}_1$ is from (A, B, C, D, E) to (`S1.suppkey`,

`S1.name`, `PS1.availqty`, `S2.suppkey`, `S2.name`). Often, a very large number of potential column mappings for R_{out} are identified by the algorithm. To address column ambiguity, we need to find the correct column mapping from among the candidate mappings.

Column Cover. See the definition in Section 2.

Walks. A walk is a sequence $v_0, e_1, v_1, \ldots, v_k$ of graph vertices v_i and graph edges e_i such that for $1 \leq i \leq k$, the edge e_i has endpoints $v_{(i-1)}$ and v_i [34].

When the algorithm chooses a promising column mapping $\mathcal{M}$ for R_{out}, it then determines the set of table instances $\mathcal{I}_{\mathcal{M}}$ that are involved in $\mathcal{M}$. For instance, for mapping $\mathcal{M}_1$ above, set $\mathcal{I}_{\mathcal{M}_1}$ is the same as the set of the projection table instances: $\{S1, S2, PS1\}$. Addressing join-path ambiguity requires connecting these table instances via the correct combination of instantiated *walks*. For Query 1 these walks are $w_1 = S1 - PS1$, $w_2 = PS1 - P - PS2 - S2$, and $w_3 = S1 - N - S2$, see Figure 3(a).

We will refer to a combination/set of walks as a *walk group*. A walk group is *connected* if it forms a connected graph; such a group corresponds to a candidate query. Hence, the task is to find the correct walk group out of very large number of possible walk groups. For Query 1 the correct walk group is $W = \{w_1, w_2, w_3\}$.

When dealing with walks the algorithm might not initially assign instances to their intermediate nodes and such walks are called *uninstantiated*, otherwise they are called *instantiated*. For instance, walk $u_2 = PS1 - P - PS - S2$ is uninstantiated walk that correspond to instantiated walk w_2. Walks w_1, w_2, and w_3 are examples of *simple walks*, *i.e.,* walks whose nodes are all distinct. Walk $S1 - PS1 - P1 - PS1 - S2$, illustrated in Figure 5(a), is an example of a non-simple walk, as it visits the instantiated node $PS1$ twice.

Problem Definition. Having introduced the notation, we now can define the two QRE variants. Let *a generating query* Q_{gen} be a query that generates R_{out} on $\mathcal{D}$, that is, $Q_{gen}(\mathcal{D}) = R_{out}$. The QRE problem is defined as:

Definition 3.1 (Exact QRE). Given database $\mathcal{D}$ with its schema graph G_S and output table R_{out}, find a *generating CPJ query* Q_{gen} that is consistent with G_S and such that $Q_{gen}(\mathcal{D}) = R_{out}$.

While the basic definition is asking to find a single query, some QRE solutions may provide an interface for the user to request to enumerate other generating queries.[2] FastQRE supports both of these versions, though we will limit our discussion to the version consistent with Definition 3.1. The order of enumeration is often determined by the query complexity $|Q|$, which traditionally is computed as query *description complexity* $|Q|_{dc}$. The smaller the number of tables and joins involved in Q, the smaller the value of $|Q|_{dc}$ should be. We will also refer to $|Q|_{dc}$ as the *query graph cost* of Q, as $|Q|_{dc}$ involves counting various elements of the query graph G_Q. For example, $|Q|_{dc}$ is often defined as $|Q|_{dc} = |V_Q|$, or $|Q|_{dc} = |E_Q|$, or $|Q|_{dc} = |V_Q| + |E_Q|$.

Some QRE approaches solve a simpler Superset QRE variant:

Definition 3.2 (Superset QRE). Given dataset $\mathcal{D}$ with its schema graph G_S and output table R_{out}, find a *generating CPJ query* Q_{gen} that is consistent with G_S and such that $Q_{gen}(\mathcal{D}) \supseteq R_{out}$.

While we focus on solving the exact variant of QRE problem, the algorithms proposed in this paper are generic and can benefit other QRE variant as well.

Efficiency Challenge. The problem of query reverse engineering is known for its efficiency challenge. This is since (1) its search space is very large; and (2) once a candidate query Q is constructed in this search space, testing if $Q(\mathcal{D}) = R_{out}$ could be very expensive as well, especially for complex queries and large databases. A successful approach for solving the problem thus should be able to address all these sources of inefficiency.

Naive Solution. Conceptually, the naive approach works by first computing the column cover for each column of R_{out}. It then enumerates column mappings that are possible according to this cover and enumerates walk groups that correspond to these column mappings. It checks each resulting candidate query Q to see if it generates the desired R_{out}.

4 FASTQRE APPROACH

In this section we first overview the `FastQRE` framework and then discuss all of its components in more detail.

4.1 Overview of FastQRE

Figure 6 presents a high-level architecture of the `FastQRE` framework. It is composed of four logical modules described below. Each module consists of one or more subcomponents, where the novel components proposed in this paper are highlighted in blue.

1. Preprocessing. First, the framework performs pre-processing of the input data. As Figure 6 suggests, this module consists of three components that deal with (a) initial parsing of data; (b) computing column cover; and (c) building database indexes. The input data might need to be first parsed so that it can be ingested by the system. For example, R_{out} table might come as an excel table that needs to be converted into a format the system understands. In turn, the column cover is computed as described in Example 2.2. If necessary, database indexes are built to speed up computations. Note that, even though these components are considered to be standard, some creative techniques are often used by QRE solutions to improve the efficiency. For example, computing the column cover would require a quadratic number of comparisons in the number of columns if done naively. To avoid comparing all pairs of columns, `FastQRE` first computes patterns formed by column values, that are then leveraged to avoid certain column comparisons.

2. Candidate Query Generation. The purpose of this module is to generate a good sequence of candidate queries. Queries in this sequence will be then processed by the *Query Validation* module to check if one of them is a generating query Q_{gen}. The closer Q_{gen} to the beginning of the sequence, the fewer candidate queries will need to be checked and the faster the framework will find Q_{gen}.

This module consists of four components. The *Direct Column Coherence* component allows to deal with column ambiguity by discovering coherent column groups and storing them as CGM tuples (Section 4.2). The *Ranking Column Mappings* component then

[2] Notice, there could be multiple different generating queries that all produce R_{out}. They often form equivalence classes: there will be 1 or more non-overlapping groups of generating queries, where each query in a group is semantically equivalent to the rest of the queries in the group.

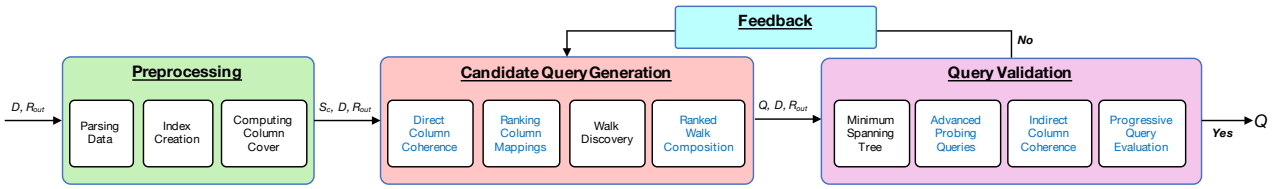


Figure 6: Architecture (Data Flow) of FastQRE. Novel components are highlighted with blue color.

uses these CGM tuples to generate a ranked sequence of column mappings (Section 4.3). Recall that resolving column ambiguity is equivalent to finding the correct column mapping. Hence, the ranking should be such that the correct column mapping should tend to be ranked higher than the other mappings.

Having chosen a column mapping to analyze, the algorithm needs to connect the table instances involved in the mapping via correct join paths. The *Walk Discovery* component discovers various walks in the schema graph that exist between the pairs of these table instances (Section 4.4). We use the standard breadth-first search algorithm to discover walks. A candidate query corresponds to a combination of such walks that connect these table instances. To generate a good sequence of candidate queries, the algorithm uses the *Ranked Walk Composition* component that considers various combinations of walks in ranked fashion (Section 4.4).

3. Query Validation. Given a candidate query Q, the task of the *Query Validation* module is to check if $Q(\mathcal{D}) = R_{out}$. Running a query on the entire database can be a computationally expensive operation, especially for a complex query on a large database. Hence, prior to doing this check, this module tries to see if the query can be dismissed quickly as the wrong query. It does it with the help of three components.

The *Advanced Probing Queries* component deserves separate thorough study; it is briefly summarized in Appendix A. The component issues specially formulated probing queries trying to find certain discrepancies that would allow it to dismiss Q. A basic probing query is based on the observation that if $Q(\mathcal{D}) = R_{out}$ then we can form a probing query Q_{prob} out of Q by adding certain conditions to Q. Those conditions should force $Q_{prob}(\mathcal{D})$ to output a single tuple t from R_{out}. The fact that $Q_{prob}(\mathcal{D}) \neq t$ would indicate that Q is not a generating query. Query Q_{prob} is constructed such that executing $Q_{prob}(\mathcal{D})$ could be much faster than executing $Q(\mathcal{D})$, resulting in a quick check. In its basic form, however, the probing query mechanism does not work well for FastQRE, see Appendix A.

The *Indirect Column Coherence* component checks for walk coherence as illustrated in Example 2.2. FastQRE employs a *lazy* implementation of this technique: walk coherence checks could be computationally expensive and thus the framework performs these checks at the very last moment. Further, it is an example of a technique that applies to a *group* of queries. That is, if a walk is not coherent, the candidate query that contains this walk and caused the check for this walk coherence will be dismissed. Furthermore, all the subsequent queries that include this walk will also either be dismissed or will not be generated in the first place (Section 4.5).

If the above two components still fail to dismiss the query, then the *Progressive Query Evaluation* component runs the check if $Q(D) = R_{out}$. However, instead of running it as a single block operation, it runs it *progressively*, using an equivalent of `getNext()` interface that gets the next result tuple, one tuple at a time. For certain wrong queries, this allows the algorithm to stop early: as soon as it finds a result tuple that contradicts R_{out}. If the check is successful, then the algorithm outputs Q as its answer.

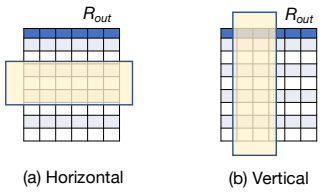


Figure 7: Horizontal and vertical checks.

4. Feedback. When the validation module dismisses the wrong candidate query Q, it propagates some useful information it computed while processing Q back to the Candidate Query Generation module using the `Feedback` module. Example of the propagated information include newly discovered non-coherent walk, the condition of why Q failed: *e.g.*, $Q(\mathcal{D}) \subset R_{out}$, or $Q(\mathcal{D}) \supset R_{out}$, and so on. The Query Generation Module uses this to generate better sequences of candidate queries.

Horizontal and Vertical Checks. It could be instructive to visualize some of the QRE techniques as horizontal and vertical checks, see Figure 7. For instance, computing column cover is an example of a vertical check which processes a single column of R_{out} at a time. The newly proposed direct and indirect coherence checks are also examples of vertical checks. However, these checks now analyze multiple columns at once using more advanced algorithms. Similarly, the mechanism of basic probing queries is an example of a horizontal check performed on a single tuple of R_{out}. The technique used by our advanced probing query component is also an example of a horizontal check. However, it also now applies to multiple entries (tuples) at once using more advanced methodology.

In the subsequent sections we explain all the FastQRE components in more detail.

4.2 Direct Column Coherence

The proposed approach employs the new concept of direct column coherence to significantly reduce the column-level ambiguity. Let C be a group (a subset) of columns from a table R and table $\pi_C(R)$ be the projection of R on columns C. Then we can define:

Definition 4.1 (Column Coherence). Column group C from R is *coherent* (with respect to columns C_{out} from R_{out}), denoted as $C_{out} \sqsubset C$, if there is a 1-to-1 mapping M that determines the correspondence among columns of C and C_{out}, such that $\pi_{C_{out}}(R_{out}) \subseteq \pi_C(R)$ according to that mapping.

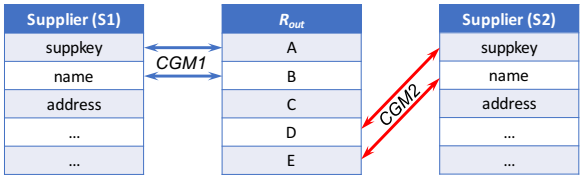


Figure 8: CGM examples: only two (of several) are shown.

For instance, recall that in Example 2.2 columns C, B of table $R1$ (see Figure 4(a)) are coherent vis-a-vis columns X, Y of R_{out} (see Figure 4(d)). The 1-to-1 mapping is $M = \{\texttt{C} \leftrightarrow \texttt{X}, \texttt{B} \leftrightarrow \texttt{Y}\}$.

Definition 4.2 (CGM). For coherent column group C, the corresponding tuple $\lambda = (R, C, M, C_{out})$ is called a *CGM*.

The term CGM is a short for terms coherency, group, and mapping. The CGM for the above example is $\lambda_1 = (R1, \{\texttt{C}, \texttt{B}\}, \{\texttt{C} \leftrightarrow \texttt{X}, \texttt{B} \leftrightarrow \texttt{Y}\}, \{\texttt{X}, \texttt{Y}\})$. Similarly, Figure 8 illustrates examples of CGM's for our running Example 2.1. The first CGM maps columns A and B of R_{out} into columns `suppkey` and `name` of the Supplier table. For this CGM, $R = \texttt{Supplier}$, $C = \{\texttt{suppkey}, \texttt{name}\}$, $M = \{\texttt{suppkey} \leftrightarrow A, \texttt{name} \leftrightarrow B\}$, and $C_{out} = \{A, B\}$. The second CGM maps columns D and E also into columns `suppkey` and `name` of `Supplier` table.

Let $C \mapsto C_{out}$ denote the fact that it is possible to construct a generating query Q_{gen} wherein columns C_{out} in R_{out} are generated from columns C from R. Informally, $C \mapsto C_{out}$ implies that it is likely that columns C have been used in the original query Q_{orig} to generate columns C_{out} in R_{out}. Then the importance of column coherence and CGMs comes from the following observations:

(1) If $C_{out} \sqsubset C$, then it *might* hold that $C \mapsto C_{out}$. This is since columns $C_{out} \subset R_{out}$ are "consistent" with columns $C \subset R$ and thus perhaps C was used by Q_{gen} to form tuples in columns C_{out}.
(2) Further, if $C_{out} \sqsubset C$, then it is *likely* that $C \mapsto C_{out}$. This is because while it is possible that $C_{out} \sqsubset C$ but $C \not\mapsto C_{out}$, in practice, it is rare that a group of columns is coherent just by chance, especially for large cardinality R_{out} and large column groups with diverse set of values.
(3) Finally, if $C_{out} \sqsubset C$, then it is likely that columns in $C \in R$ came from the same *instance* of R.

We will see that this intuition is indeed correct and works very well when we study our approach empirically in Section 5.

For a table R_i we can construct the set Λ_i of all its maximal CGMs. Intuitively, a maximal CGM is a CGM that cannot be further enlarged by adding to it another column from R_i. We will say that CGM $\lambda = (R, C, M, C_{out})$ is a subset of another CGM $\lambda = (R', C', M', C'_{out})$, if $R = R'$, $C \subset C'$, $C_{out} \subset C'_{out}$, and 1-to-1 mapping M is consistent with M', that is, it maps columns $C \leftrightarrow C_{out}$ identically to the mapping M' for these columns. Now we can define:

Definition 4.3 (Maximal CGM). A CGM λ is *maximal* and belongs to Λ_i if λ is not a subset of any other CGM for R_i.

Notice, any proper subset of $\lambda \in \Lambda_i$ is also a CGM, but, by definition, does *not* belong to Λ_i. In addition, observe that if two CGMs $\lambda_1, \lambda_2 \in \Lambda_i$ are part of Q, then they cannot be part of the same *instance* of R_i in a generating query. This is because, otherwise, a single CGM $\lambda_1 \cup \lambda_2$ would have been part of Λ_i. This point is highlighted in Figure 8, where two distinct instances S1 and S2 of the Supplier table are used to illustrate two maximal CGMs: CGM1 and CGM2. In subsequent discussions when we talk about CGMs we will always assume maximal CGMs unless stated otherwise.

In Figure 8, the CGM that corresponds to mapping $\{\text{suppkey} \leftrightarrow A\}$ is not maximal, because it is part of a larger CGM1 with mapping $M = \{\texttt{suppkey} \leftrightarrow A, \texttt{name} \leftrightarrow B\}$. Figure 8 does not show it, but CGM1 and CGM2 are maximal as they cannot be enlarged.

4.3 Ranking Column Mappings

We now will consider various properties of CGMs that can be employed to rank the various column mappings. After that we will discuss the ranking algorithm that leverages these properties with the goal of assigning the higher score to the correct mapping.

4.3.1 Properties of CGMs. CGMs have several important properties that can be utilized to address column-level ambiguity of the search space and to rank the various column mappings. Recall that, if a CGM involves a large number of columns, then there is certain likelihood that such a relationship among columns is not by chance and that this CGM has been used in the original query. In practice, this likelihood is very high. Let us define:

Definition 4.4 ($\lambda \in Q$). A given CGM $\lambda = (R, C, M, C_{out})$ is *part of query* Q (or, Q *uses* CGM λ), denoted $\lambda \in Q$, if Q uses columns C to generate columns C_{out} in R_{out} consistently with the mapping M and all columns in C come from the same instance of table R.

Similar to computing the column cover S_c for each column $c \in R_{out}$, we can also compute the set Λ_c of all the (maximal) CGMs that column c is part of. Now assume that for some column $c \in R_{out}$ it holds that $|S_c| = 1$ and $|\Lambda_c| = 1$. This means that c is a 1-match column: as c maps only into a single column $S_c = \{c_1\}$ and a single CGM $\Lambda_c = \{\lambda\}$, where $\lambda = (R, C, M, C_{out})$. This case is frequent in practice and can occur for several columns in R_{out}. For this case we know that c_1 *must* be part of (the `SELECT` clause of) any generating query Q_{gen} in the context of some instance R^k of R. Because c_1 is part of λ, chances are that: (a) columns in C are also part of query Q_{gen}; (b) they are present in the context of the same instance R^k of R; and (c) that they are used to generate columns C_{out} of R_{out}. We can very effectively leverage this observation to address column-level ambiguity by preferring some column mappings to others. Furthermore, it is possible to show that when column $c' \in R$ that corresponds to 1-match column c is a key column in $\pi_C(R)$, then we can safely assume that Q uses CGM λ.

For example, let us consider R_{out} for Query 1. Its column A maps into five CGMs, column C into four CGMs, and column D into five CGMs. However, its column B maps only to CGM1 and column E only to CGM2 illustrated in Figure 8.

Notice how this technique correctly located 2 out of 3 projection table instances $S1$ and $S2$ as well as 4 out of 5 projection columns involved `S1.suppkey`, `S1.name`, `S2.suppkey`, and `S2.name`. At this stage the algorithm knows only that columns A, B, D, and E *possibly* have been generated from these 4 columns. After factoring in an additional fact that `S.name` uniquely determines column `S.suppkey` the algorithm can guarantee that CGM1 and CGM2 are part of Q_{gen}.

4.3.2 Using CGMs for Ranking. For each table R_i its set of maximal CGMs Λ_i can be computed using approaches that are similar to finding association rules and functional dependencies [1, 23], as they discover consistency of values in multiple columns. Once CGMs are computed, they are used by the algorithm for constructing column mappings in a ranked order as explained below.

Certain Column Assignments. The algorithm starts the construction by making assignments for 1-match columns from R_{out} because they are certain. It then adds to them columns for which these 1-match columns act as keys, as described in Section 4.3.1. This process could result in assigning all columns of R_{out}, in which case the algorithm stops. Otherwise, the algorithm proceeds to the next step. As we know, for Query 1 this step results in determining the mapping for 4 out 5 columns of R_{out}.

Uncertain Column Assignments. The algorithm then considers each unassigned column and enumerates its possible assignments. It leverages CGMs in pruning certain combinations of column assignments. Namely, to check if a group of columns $C' \subseteq R_i$ can be assigned to the same instance of table R_i, the algorithm checks if a CGM $\lambda = (R, C, M, C_{out})$ exists in Λ_i such that $C' \sqsubset C$. If it does not, then, by definition, C' cannot be coherent and thus cannot be assigned to the same instance of R_i.

Ordering Assignments. The algorithm uses two criteria to decide the order in which column assignments are considered. The first criterion is minimizing the overall number of projection table instances in an assignment. Ties are broken by considering the second criterion, which computes the score for each column assignment. The score is based on the Jaccard similarity between the column value sets.

For Query 1, the only uncertain column that will remain for R_{out} is column C. It can map into 4 CGM's that map C into (1) `C.custkey`, (2) `P.partkey`, (3) `PS.partkey`, and (4) `PS.availqty`. Option 4 (the correct one in this case) wins as having the largest Jaccard similarity score of 1. Hence, the algorithm will consider this option first.

This overall ranking strategy has been found to be very effective. The correct column mapping that we need to find is always present among the first few top-ranked mappings suggested by this strategy.

4.4 Ranked Walk Composition

Given a column mapping $\mathcal{M}$, this component will analyze the set of table instances $\mathcal{I}_{\mathcal{M}}$ that are involved in this mapping. To address the join-path level ambiguity, it will need to interconnect these instances via correct join paths. For this task, it first discovers the set W of all L-short walks between pairs of these instances. It then will need to enumerate, in a ranked order, over different combinations of these walks. Since a connected walk combination/group corresponds to a candidate query, this component essentially enumerates candidate queries in a ranked order. The Query Validation module will later test these candidates to find a generating query.

In this section we first present a basic approach for generating walk groups. We then analyze its drawbacks and present an improved solution that addresses those drawbacks.

4.4.1 Basic Approach. First, the basic approach generates the set of all L-short walks W. Each walk starts and ends with an instance

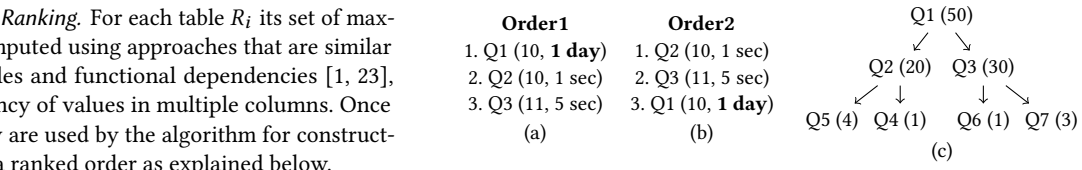


Figure 9: Illustration of the drawbacks.

from $\mathcal{I}_{\mathcal{M}}$, but does not have any instances from $\mathcal{I}_{\mathcal{M}}$ as intermediate nodes. To generate candidate queries the algorithm should be able to enumerate all the subsets of W. The number of subsets can be large: $O(2^{|W|})$, where $|W|$ can be above 100. Thus, the algorithm should avoid generating repeated subsets for efficiency. It also should generate these walk groups in a rank order based on how likely they are to correspond to a generating query.

Hence, a natural solution is a bottom-up approach that generates candidate queries in the order of their complexity $|Q|_{dc}$. A basic approach thus maintains a priority queue PQ for generating and storing candidate queries, ordered by $|Q|_{dc}$, where we compute $|Q|_{dc}$ as the sum of the walk lengths that query Q is composed of, that is, $|Q|_{dc} = \sum_{w \in Q} |w|$.

The PQ is first initiated by adding $|W|$ queries corresponding to each single walk $w_i \in W$ to it. Then, the best cost query Q is retrieved from PQ and checked if its G_Q is connected, that is, if all the tables instances in $\mathcal{I}_{\mathcal{M}}$ are interconnected by the walks in the walk group for Q. If G_Q is connected, then Q is passed to the Query Validation module to check if it is Q_{gen}.

In case $Q \neq Q_{gen}$, the algorithm would then create sub-subqueries of Q; here, Q is a *parent* query and its subqueries are its *children*. The algorithm adds subqueries of Q to PQ as follows. In general, any query Q corresponds to a set of walks from W, *e.g.*, $\{w_5, w_{12}, w_{20}\}$. To avoid generating repeated subsets of W, the algorithm finds in Q the walk with the lowest index: $k = \min\{i : w_i \in Q\}$, *e.g.*, for $\{w_5, w_{12}, w_{20}\}$ $k = 5$. It then generates $k - 1$ sub-queries as $Q_i = Q \cup \{w_i\}$, for $i = 1, 2, \ldots, k-1$. This way, all subsets of W will be enumerated without repetitions and candidate queries are considered in the order of their complexity $|Q|_{dc}$.

Drawbacks. The above basic solution, however, suffers from two major drawbacks. First, using query description complexity $|Q|_{dc}$ alone is often suboptimal. It can lead to the *convoy effect*: the cases where concise but very long running candidate queries are evaluated prior to fast-running queries, resulting in very poor response time for finding a generating query.

For example, consider Order1 of queries $Q1, Q2, Q3$ in Figure 9(a). The notation $Q1(10, 1day)$ means $|Q1|_{dc} = 10$ and Q1 needs 1 day to complete. Let t be the response time of the algorithm needed to find Q_{gen}. Then, for Order1, regardless of which of the queries is Q_{gen}, t is at least 1 day. Figure 9(b) shows Order2 of these queries. It is often a better order as it improves the average response time: if $Q2 = Q_{gen}$ then t is only 1 sec.; if $Q3 = Q_{gen}$, then t is $1 + 5 = 6$ secs. If $Q1 = Q_{gen}$, then t is 1 day and 6 secs.

The second drawback is that, due to the way the basic approach generates queries (and regardless of the cost function used), parent queries are always tested prior to their children and further descendants. This creates a problem, as even if we use an oracle scoring function $|Q|^*$ that perfectly pinpoints the right generating query Q_{gen} out of all candidate queries, this Q_{gen} will not be present in PQ until all of its ancestors are tested. Further, its ancestors might

have poor scores[3], leading to the basic approach going over a large number of wrong candidate queries prior to reaching the right one.

Figure 9 (c) illustrates an example of a query generating tree: queries Q2 and Q3 are generated out of $Q1$, and so on. The 50 in notation $Q1(50)$ means the cost of query Q1 is 50 according to some (good) cost metric. The above approach will be forced to test $Q1$ prior to testing all of its descendants, whereas the cost function suggests trying $Q4$ or $Q6$ first as they have the smallest costs of 1.

4.4.2 Improved Approach. To address the two drawbacks of the basic approach we propose a solution that is based on two priority queues PQ_1 and PQ_2 and two cost functions: $|Q|_{dc}$ and $|Q|_{ex}$. The first function $|Q|_{dc}$ reflects Q's description complexity and is based on the complexity of Q's query graph. The second function $|Q|_{ex}$ is based on Q's predicted *execution time* which we get from the DBMS's query optimizer. *To the best of our knowledge, $|Q|_{ex}$ has never been used in the past for solving the QRE problem.*

Neither of these two cost functions is perfect when used alone. For example, $|Q|_{dc}$ alone can choose concise but very long-running queries. This is a problem since to reduce the average expected response time, equal queries should be run in the ascending order of their execution cost; otherwise, the response time can suffer very significantly. Similarly, using $|Q|_{ex}$ alone as a metric could lead to various problems. This happens for various reasons, including the query optimizer not always being able to accurately predict the query execution time. As a result, $|Q|_{ex}$ metric alone can prefer, say, a candidate query that joins 12 tables to a query that joins only 3 tables, as the optimizer might decide that the 12-table query is slightly faster to execute.

Hence, our solution combines these two cost functions to form a new cost function $|Q|_\alpha = \alpha|Q|_{dc} + (1-\alpha)|Q|_{ex}$, where $\alpha \in [0,1]$ determines the contribution of each cost.[4] The value of α is set in a semi-automated fashion as follows. Given a database and its schema, either the analyst, or the QRE approach itself, generates a few test queries and their corresponding R_{out} tables. Tests then are done to determine which α results in good performance for the test queries.

Algorithm. Algorithm 1 describes our solution. It assumes the set W is already generated the same way as in the basic approach. It uses two priority queues PQ_1 and PQ_2, where PQ_1 orders candidate queries based on $|Q|_{dc}$ metric, whereas PQ_2 uses $|Q|_\alpha$. The algorithm starts by initializing PQ_1 with queries that correspond to each single discovered walk from W (Lines 1 and 2). Then, while PQ_1 is not empty, it repeatedly extracts the next best query from PQ_1 according to $|Q|_{dc}$ (Lines 3 and 4). The algorithm then adds child sub-queries of Q to PQ_1: in the same way that avoids repetitions as have been described for the basic approach (Lines 5 – 8).

Next, a check is done whether G_Q of Q is connected (Line 9). If not, Q cannot be a generating query and the algorithm skips Q and returns back to the first `while` loop. Otherwise, Q might be a generating query and thus the algorithm inserts Q in PQ_2 (Line 10) and proceeds forward to the second `while` loop (Line 12).

[3]One reason for that is that those queries are missing the right walks, which correspond to additional restricting conditions. Their absence can lead to large result sets that are costly to compute.

[4]The actual combining function can also be chosen differently from this method, as long as it balances the query execution cost and its description complexity.

Algorithm 1: Ranked Walk Composition

Input: $\mathcal{D}$, R_{out}, W
Output: Q: generating query for R_{out}

1 **foreach walk** $w_i \in W$ **do** // Init PQ_1
2 $PQ_1.push(\{w_i\})$
3 **while** $|PQ_1| > 0$ **do**
4 $Q \leftarrow PQ_1.pop()$
5 $k \leftarrow \min\{i : w_i \in Q\}$
7 **for** $i \leftarrow 1, 2, \ldots, k-1$ **do**
8 $PQ_1.push(Q \cup \{w_i\})$
9 **if** Is-Connected(G_Q) = `false` **then continue**
10 $PQ_2.push(Q)$
12 **while** $|PQ_2| > 0$ **do**
13 **if** $|PQ_1| > 0$ & $|PQ_1.peek()|_{dc} \leq |PQ_2.peek()|_{dc} + C_1$ & $|PQ_2| < C_2$ **then break**
14 $Q \leftarrow PQ_2.pop()$
16 **if** Validate-Query(Q) **then return** Q
17 **return** $\emptyset$

The second `while` loop iterates until PQ_2 is not empty. Inside this loop, the algorithm first tries to break out of the loop by checking three conditions (Line 13). The first condition checks whether PQ_1 is empty, since if it is, all the remaining candidate queries are stored only in PQ_2 and thus the algorithm should not break from the second loop. The second condition compares the top/best elements (*i.e.,* candidate queries) of PQ_1 and PQ_2. If PQ_1 still has a "good" candidate whose $|Q|_{dc}$ score is not far from the $|Q|_{dc}$ score of PQ_2 the algorithm will attempt to break after checking the third condition. This second condition ensures that PQ_2 stores a certain *pool* of candidate queries with good $|Q|_{dc}$ scores, out of which the algorithm will be able to select the best query in terms of $|Q|_\alpha$ score. The third condition controls the size of this pool: if it already has a large number of candidate queries to consider, the algorithm will not break from the second `while` loop.

If the algorithm does not break from the second loop, it retrieves the best candidate query Q from PQ_2 (Line 14) and passes it to the Query Validation module. If that module returns that $Q = Q_{gen}$ then the approach outputs Q and stops, otherwise it will continue the second loop. In case the approach cannot find the generating query, it will terminate and return $\emptyset$.

Notice how this algorithm will easily handle the two drawbacks illustrated in Figure 9. The cost function $|Q|_\alpha = \alpha|Q|_{dc} + (1-\alpha)|Q|_{ex}$ will handle the drawback shown in Figure 9(a). Any reasonable value of α will result in reordering Order1 (Figure 9(a)) into Order2 (Figure 9(b)). For the drawback in Figure 9(c), the created query pool will allow the algorithm to look at all the queries $Q1, Q2, \ldots, Q7$ at once, and pick $Q4$ or $Q6$ first: as having the lowest cost.

4.5 Query Validation

Given a candidate query Q, the task of the Query Validation module is to check if $Q(\mathcal{D}) = R_{out}$. Since this check can be expensive, the approach first tries several methods to quickly dismiss query

Q without performing this check. If it cannot, it will then test if $Q(\mathcal{D}) = R_{out}$, progressively. That is, it will use an analog of `getNext()` interface provided by most of the modern DBMS's to retrieve $Q(\mathcal{D})$ results one tuple at a time and see if the results returned so far fully agree with R_{out}. This way the algorithm has the opportunity to stop early if the candidate query is wrong, without executing the entire query Q on $\mathcal{D}$ as a block operation. Frequently, the algorithm stops after very few calls to `getNext()`.

When trying to dismiss Q, the approach first performs an optional check if Q forms the only minimum spanning tree (MST), in which case it skips the rest of the steps and proceeds directly to evaluating if $Q(\mathcal{D}) = R_{out}$ progressively. This MST optimization, if present, makes the approach always perform no worse than a naive approach of always connecting the projection tables via MST, without applying the subsequent steps that might require some time to compute. That naive solution, while applicable to a narrow class of queries (only those that are connected via the MST), is fast at discovering those queries. Hence, this optional step might be desirable when many queries to reverse engineer are MST queries.

As the next step the algorithm invokes the Advanced Probing Query component on Q, as summarized in Appendix A. It works by forming probing queries out of Q and checking for consistency of their results. If it cannot dismiss Q, the algorithm invokes the indirect column coherence component.

Indirect Column Coherence. Similar to Definition 4.1 that defines (direct) column group coherence, we can also define (indirect) column group coherence with respect to a walk, which we also will refer to as *walk coherence*. Let $\lambda_1 = (R_i, C_1, M_1, C_1^{out})$ and $\lambda_2 = (R_j, C_2, M_2, C_2^{out})$ be two CGMs and w be a $\lambda_1 \rightsquigarrow \lambda_2$ walk having these two CGMs as its end points. In a query this walk corresponds to a join, whose resulting relation we will refer to as R_w. Let us define $C = C_1 \cup C_2$, $C_{out} = C_1^{out} \cup C_2^{out}$, and $M = M_1 \cup M_2$ which is a 1-to-1 mapping that maps columns in C and C_{out}.

Definition 4.5 (Walk Coherence). Walk w is *coherent* (or, alternatively, C and C_{out} are coherent with respect to w) if $\pi_{C_{out}}(R_{out}) \subseteq \pi_C(R_w)$ where columns are mapped according to M.

The significance of the notion of walk coherence comes from the following important lemma:

Lemma 4.6 (Walk Coherence). *In a generating query,* all *of its walks* must *be coherent.* □

Hence, the algorithm checks for walk coherence of Q. In general, such a check involves scanning and joining tables and thus could be a relatively expensive operation. To perform this check efficiently, the algorithm uses three different techniques.

First, the approach does not check coherence of walks right after these walks are discovered. Instead, it does it in a lazy fashion: the coherence is checked only at the last moment when it is needed. Checking for coherence right away can reduce the number of candidate queries put into PQ_1, but will incur the cost of all the checks for each walk in W. The lazy check proves to be significantly more efficient, as performing all the walk checks requires querying $\mathcal{D}$, whereas generating candidate queries does not involve querying $\mathcal{D}$ and as such it is very efficient, whereas the wrong queries are still successfully pruned away by the lazy check later on.

Second, when a walk is checked for coherence, the outcome is recorded for that walk and never recomputed again. This helps avoid re-computations as multiple distinct queries can share the same walk. To check a query for coherence, the algorithm first scans through the walks in the query whose status has already been determined, trying to find an incoherent walk. If it succeeds, it filters away the query – without running any new walk coherence checks. Otherwise, it scans through the remaining walks one by one, running walk coherence checks. If it finds an incoherent walk, it stops immediately without checking the remaining walks and filters Q away.

The third method is based on the intuition that when a walk is incoherent, this often reveals itself relatively quickly, after checking a few tuples from R_{out}. However, when a walk is coherent, the check runs for longer time needed to test each tuple in R_{out}. This observation is used by the algorithm which has the option to not run the full coherence check to completion, but stop early based on some criteria, such as a timeout or a certain *sample* of R_{out} being verified. If the walk is not coherent, that is often still successfully detected by this method, prior to the timeout. If the algorithm cannot detect walk incoherence by the timeout, w is probably coherent, but the algorithm does not know that with certainty. Thus, the algorithm then treats the walk as if it is coherent, which is safe as the query is not dismissed. This methodology significantly speeds up the average time needed for walk coherence checks.

5 EXPERIMENTAL EVALUATION

In this section we empirically evaluate our approach. The experiments have been run on a machine with 2.8 GHz Core i7 CPU and 16 GB of RAM: on a single core and a single thread.

Experimental Setup. The experiments have been conducted on the TPC-H benchmark dataset [29]. We use two different data generators to populate the TPC-H database:

(1) `TPCH1` dataset (126 MB). TPC-H database generated by Microsoft Research (MSR) data generator [20]. We use this dataset to compare `FastQRE` to [38], using skewed data distributions.
(2) `TPCH2` dataset (1.1 GB). This is the original TPC-H dataset generated with the original TPC-H data generator [29]. Hence, we use this dataset to test `FastQRE` on the original TPC-H.

Even though `TPCH1` and `TPCH2` have the same TPC-H schema, they have different value compositions and `FastQRE` behaves quite differently on them in many respects.

We consider the 21 queries TQ1, TQ2, . . ., TQ21 from [38]. We have contacted the authors for the additional information on the queries. The queries have been derived from the 21 TPC-H queries. TQ22 is the only query from [38] where our approach does not apply, as it contains a small non-simple instantiated walk, and, hence, it is not used in our experiments.

Background on the Star system. We will compare the performances of `FastQRE` and a state of the art technique [38], which we will refer to as `Star`. `Star` works for queries that involve at least one join. It is also very memory intensive and hence [38] tests it on 128 GB RAM. In our setup with 16 GB RAM, `Star` simply runs out of memory for many queries, producing meaningful results only for 6 queries: TQ4, TQ11, TQ12, TQ13, TQ14, and TQ17. This shows another advantage of `FastQRE` over `Star`: it is not only faster, but can run many more TPC-H queries with a smaller RAM footprint.

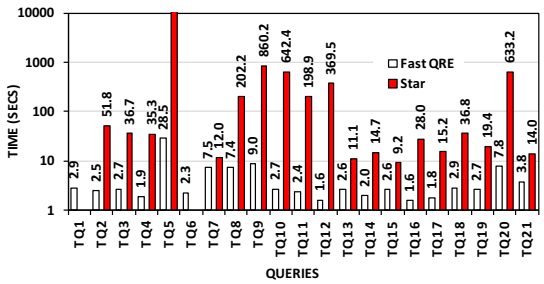


Figure 10: Comparing execution time (log scale).

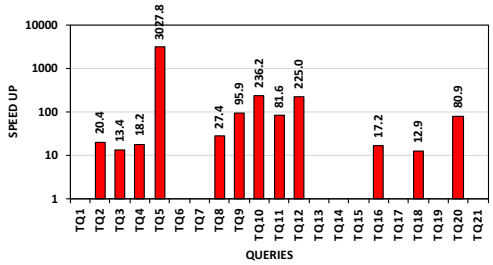


Figure 11: Speedup of `FastQRE` over `Star` (log scale).

Experiment 5 in Appendix B compares the old Star results from [38] for 128 GB machine to the new Star results that we get for the 6 queries on our 16 GB machine. It shows that the new results are actually slower on our machine for 4 out of 6 queries. They are faster for 2 out of 6 queries, but by no more than 24%.

Experiment 5 also compares the new results of `Star` to those of `FastQRE` on these 6 queries, showing that `FastQRE` is 1-2 orders of magnitude faster on the same hardware. However, to get a broader picture of the performance, it is interesting to have such a comparison on more than 6 queries. Given that the new Star results are only at most 24% faster than the old results, we next present such a comparison to the results of `Star` reported in [38].

Experiment 1 (Efficiency of `FastQRE` and `Star`). In this experiment we use `TPCH1` dataset to compare the results of `FastQRE` and the results of `Star` reported in [38]. Figure 10 shows the running time of the two algorithms in seconds. For both techniques this cost excludes the cost of running the final $Q(\mathcal{D}) = R_{out}$ tests, whereas its contribution is studied separately in Experiment 2. The filled bar corresponds to the results of `Star` reported in [38], whereas the empty bars correspond to `FastQRE`. The labels on top of bars correspond to the actual running time in seconds.

In Figure 10 `Star` demonstrates reasonable performance on 14 of 21 queries. However, the graph shows large spikes in processing for 7 of 21 complex queries: TQ5, TQ8, TQ9, TQ10, TQ11, TQ12, and TQ20. For example, the difference in processing time for `Star` for TQ9 and TQ15 is almost 2 orders of magnitude. In contrast, `FastQRE` shows results that look more uniform and do not have large spikes. For example, the difference in performance between TQ9 and TQ15 is less than 1 order of magnitude. `FastQRE` is much faster to process the 7 queries that are challenging for `Star`, which allows the analyst to save a lot time on the QRE process.

The worst performing query for `Star` is TQ5 which it could not resolve in 1 day and it had to be stopped. By design, `Star` will reverse engineer TQ5 *eventually*, but its machinery is not effective enough to do it in a reasonable amount of time. The worst query for `FastQRE` is also TQ5, but it takes only 28.5 seconds to resolve, which is at least 3 orders of magnitude faster than `Star`.

Module/Component	Time `TPCH1`	Time `TPCH2`
Reading Data	12%	4%
Computing Column Cover	6%	3%
Direct Column Coherence	3%	8%
Rest of Candidate Query Generation	42%	16%
Indirect Column Coherence	1%	4%
Advanced Probing Queries	3%	1%
Final Progressive Check	34%	64%

Table 2: Relative composition of phases.

Figure 11 illustrates the speedup achieved by `FastQRE` over `Star` for the cases where the speedup was at least 1 order of magnitude.[5] We can see that for the 7 challenging queries, the median speed up achieved by `FastQRE` is about 2 orders of magnitude.

Experiment 2 (Relative composition of phases). Table 2 presents the relative composition of the execution time for the various component of the `FastQRE` framework for `TPCH1` and `TPCH2` datasets, see Section 4.1. We will discuss the results for `TPCH2` separately in Experiment 6.

For `TPCH1`, the first (preprocessing) phase takes only 18% of the overall end-to-end running time of the algorithm. It consists of reading the data (12%) and computing column cover (6%). The framework then spends 3% handling direct column coherence and 42% on the rest of the Candidate Query Generation. Only 1% is spent on Indirect Column Coherence and 3% on Advanced Probing Queries: this is a good result as these components supposed to perform their checks quickly. The final $Q(\mathcal{D}) = R_{out}$ progressive check takes 34%. This indicates that the main logic of `FastQRE` is very efficient when compared to the time needed to compute $Q(\mathcal{D})$.

Experiment 3 (Quality of FastQRE). `Star` approach is theoretically capable of resolving each of 22 TPC-H queries. However, it runs out of memory in our 16 GB setup for many of the queries, and is able to handle only 6 out of 22 queries in the end. Hence, its effective accuracy is $6/22 = 27.3\%$. The effective accuracy of FastQRE is 21/22, as it cannot handle TQ22 which contains a non-simple walk.

We next study the quality of the Candidate Query Generation (CQG) module. Recall that CQG is the second module in the framework, see Figure 6 in Section 4.1. At a high level, its task is to generate a sequence of candidate queries to test: to check if they are Q_{gen}. For example, it generates first candidate query CQ_1, then the validation module checks if it is Q_{gen}. If not, CQG module will generate the second candidate query CQ_2, and so on. This process continues until CQ_n is found that is equal to Q_{gen}, at which point query $CQ_n = Q_{gen}$ will be presented to the user. Hence, the best case for CQG module, and for the overall framework, is when $n = 1$. That is, the ideal case is when the very first candidate query it generates is Q_{gen}. In contrast, very large values of n indicate poor quality sequences.

Figure 12 plots these values of n for different queries. Notice, for 17 out of 21 queries it holds that $n = 1$ and the generating query is

[5] `FastQRE` has shown better results for the rest of the cases as well, but we will treat the difference as insignificant.

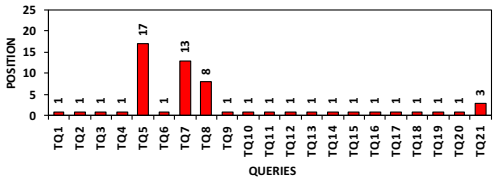


Figure 12: Position of Generating Query.

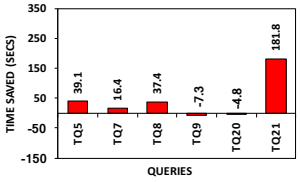


Figure 13: Time saved

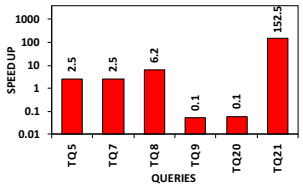


Figure 14: Filtering: Speedup

the first candidate query tried. The four queries where that is not the case are TQ5 (position is 17), TQ7 (13), TQ8 (8), and TQ21 (3). This shows the high quality of the CQG module and its subcomponents used by FastQRE. It also shows that the CQG module is a crucial part for achieving the overall good FastQRE results.

Experiment 4 (Effectiveness of Query Validation). In this experiment we examine the combined effect of the Query Validation components: Advanced Probing Queries, Indirect Column Coherence, and MST optimization.

Let t_{on} (t_{off}) be the running time of Algorithm 1 with all these components switched on (off), without the time needed for the final $Q(\mathcal{D}) = R_{out}$ check. Figure 13 plots the saved time (*i.e.,* $t_{off} - t_{on}$) and Figure 14 plots the speed up (*i.e.,* t_{off}/t_{on}) achieved by using these components. They plot the results only for the queries that have at least 5% difference in their results with filtering on vs. off.

Case 1: $CQ_1 = Q_{gen}$. The expectation is that these three components should *not* help cases where $CQ_1 = Q_{gen}$. This is confirmed in the figures. The components do not change the results by more than 5% for 15 out of 21 queries. For some queries the result could drop and we see that effect for two queries TQ9 and TQ20. This is the effect of the components running for longer time between TQ9 and TQ20, but not being able to dismiss CQ_1 since $CQ_1 = Q_{gen}$.

Case 2: $CQ_1 \neq Q_{gen}$. The three components are expected to help best when $CQ_1 \neq Q_{gen}$, but instead where Q_{gen} is not among the first few candidate queries. We see this very effect in the figures, which show the improvement for the same four queries TQ5, TQ7, TQ8, and TQ21 from Experiment 3. The biggest improvement is for query TQ21: 181 secs which corresponds to the speed up of 152 times. The reasons for it is that for TQ21, its generating query Q_{gen} is the third candidate query $CQ_3 = Q_{gen}$. When the three components are on, they successfully dismiss both CQ_1 and CQ_2, which are very expensive in terms of their execution cost. When the filters are off, CQ_1 and CQ_2 are evaluated resulting in significantly worse performance compared to the case with filters on.

Overall, having the three components on has a smoothing effect, where the performance of simple case (Case 1) queries does not change much or drops somewhat for a few queries, but the performance of complex case (Case 2) queries can improve dramatically.

6 RELATED WORK

Many research efforts studied in the literature are relevant to the QRE task, e.g. [2, 4, 7–10, 12–19, 24–26, 30, 31, 35–37, 39, 40]. We summarize the most related work below.

Query Class. The class of queries that a QRE solution can handle also determines the complexity of the problem. For instance, solving QRE for queries with arbitrary arithmetic expressions in the joins is known to be PSPACE-Hard [30]. Most of the existing approaches, including our solution, consider QRE problems for a subclass of project-join SQL queries without arithmetic expressions, many of those problems are known to be NP-Hard [27, 38]. Techniques also exist that are designed for Top-K queries [22], or focus on dealing with groupby/aggregation and unions [28, 31] in SQL queries. Our FastQRE solution can handle all CPJ queries, see Section 3.

QRE Variants. Wang et al. [32] describe an approach to solve the exact QRE problem for a rich set of SQL queries on small databases (fewer than 100 cells each). This approach enumerates abstract SQL queries in increasing order of description complexity. However, such enumeration-based techniques do not scale to large databases, which is the focus of this paper. We have already discussed the exact and superset variants of QRE. The superset QRE task has a sub-variant where the user specifies R^{+}_{out} as a table with very few (*e.g.,* 4) positive example tuples that the output should contain, *e.g.,* [27]. In another variant, the user in addition specifies R^{-}_{out} that stores negative examples that the output should not contain, *e.g.,* [5, 6, 33]. In particular, Weiss and Cohen [33] investigate the computational complexity of learning SPJ queries from positive and negative examples. Both of these QRE sub-variants can be solved using probing queries, see Appendix A. However, this method will not work well for the exact version of QRE, as issuing a probing query per each tuple in (a large) R_{out} may take months to finish.

Research efforts like [8, 22, 30] solve another QRE problem. Given a candidate query Q over a database, their task is to find the right *selection conditions* for Q such that $Q(\mathcal{D}) = R_{out}$. With the help of such techniques, FastQRE can be made to handle general SPJ queries with selection conditions, not only project-join queries.

Schema Mapping. Schema mapping work is also related, *e.g.,* [3, 11, 21]. In Clio [21], the analyst provides specifications for transforming values from input tables/columns into target tables/columns. Clio finds most likely SQL queries for the transformation. In [3], the user specifies examples of tuple values, and the system attempts to suggest transformation rules which can be edited by the user. In contrast to these approaches, QRE solutions cannot rely on enumerating large number of candidate queries, as testing even a single candidate query can be computationally expensive.

7 CONCLUSIONS

We presented the `FastQRE` approach for solving the problem of query reverse engineering. The solution gains its efficiency by leveraging novel techniques to address column-level and join path level ambiguity, by analyzing column values. An extensive empirical evaluation demonstrates the advantages of the proposed solution, which outperforms the state of the art approach by as much as 2-3 orders of magnitude. As our future work we plan to look into applying the coherence techniques for data lineage tracking.

REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, 1993.
[2] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.
[3] B. Alexe, B. ten Cate, P. G. Kolaitis, and W. C. Tan. Designing and refining schema mappings via data examples. In *SIGMOD*, 2011.
[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.
[5] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive inference of join queries. In *EDBT*, 2014.
[6] A. Bonifati, R. Ciucanu, and S. Staworko. Learning join queries from user examples. *ACM TODS*, 40(4), 2016.
[7] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. *PVLDB*, 1(1), 2008.
[8] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. In *ICDT*, 2010.
[9] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *SIGMOD*, 2002.
[10] G. J. Fakas, Z. Cai, and N. Mamoulis. Size-l object summaries for relational keyword search. *PVLDB*, 5(3), 2011.
[11] G. Gottlob and P. Senellart. Schema mapping discovery from data instances. *J. ACM*, 57(2), 2010.
[12] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: Ranked keyword searches on graphs. In *SIGMOD*, 2007.
[13] V. Hristidis, H. Hwang, and Y. Papakonstantinou. Authority-based keyword search in databases. *TODS*, 33(1), 2008.
[14] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.
[15] H. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. *SIGMOD*, 2007.
[16] M. Jayapandian and H. V. Jagadish. Automated creation of a forms-based database query interface. *PVLDB*, 1(1), 2008.
[17] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: An effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, 2008.
[18] H. Li, C. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. *PVLDB*, 8(13), 2015.
[19] A. Meliou, W. Gatterbauer, and D. Suciu. Reverse data management. *PVLDB*, 4(12), 2011.
[20] Microsoft Research. Data generator. ftp://ftp.research.microsoft.com/users/viveknar/TPCDSkew/.
[21] R. Miller, L. Haas, and M. Hernandez. Schema mapping as query discovery. In *VLDB*, 1999.
[22] K. Panev and S. Michel. Reverse engineering top-k database queries with PALEO. In *EDBT*, 2016.
[23] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *SIGMOD*, 2016.
[24] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD*, 2012.
[25] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: The power of rdbms. In *SIGMOD*, 2009.
[26] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Querying communities in relational databases. In *ICDE*, 2009.
[27] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. In *SIGMOD*, 2014.
[28] W. C. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava. Reverse engineering aggregation queries. In *VLDB*, 2017.
[29] TPC. TPC benchmarks. http://www.tpc.org/.
[30] Q. T. Tran, C. Chan, and S. Parthasarathy. Query by output. In *SIGMOD*, 2009.
[31] Q. T. Tran, C. Y. Chan, and S. Parthasarathy. Query reverse engineering. *VLDB J.*, 23(5), 2014.
[32] C. Wang, A. Cheung, and R. Bodík. Synthesizing highly expressive SQL queries from input-output examples. In *PLDI*, 2017.
[33] Y. Y. Weiss and S. Cohen. Reverse engineering spj-queries from examples. In *PODS*, 2017.
[34] D. B. West. *Introduction to Graph Theory*. Prentice Hall, 2 edition, 2000.
[35] X. Yang, C. M. Procopiuc, and D. Srivastava. Summary graphs for relational database schemas. *PVLDB*, 4(11), 2011.
[36] C. Yu and H. V. Jagadish. Schema summarization. In *VLDB*, 2006.
[37] C. Yu and H. V. Jagadish. Querying complex structured databases. *VLDB*, 2007.
[38] M. Zhang, H. Elmeleegy, C. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD*, 2013.
[39] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. On multi-column foreign key discovery. *PVLDB*, 3(1), 2010.
[40] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. Automatic discovery of attributes in relational databases. In *SIGMOD*, 2011.

APPENDIX

A ADVANCED PROBING QUERIES

In this section we briefly summarize our technique of using advanced probing queries. It provides a powerful mechanism for filtering away certain candidate queries that helps to avoid the very costly $Q(\mathcal{D}) = R_{out}$ checks and thus improve the overall efficiency.

Notice, while our algorithm attempts to process $Q(\mathcal{D})$ query progressively, many modern DBMS's are not optimized for progressive query execution, but rather aim to optimize the end-to-end (bulk) query cost. As a result, `getNext()` operation might sometimes behave not as progressive operation, but almost as a blocking call. That is, periodically the algorithm might be blocked waiting for an extended period of time for the first call to `getNext()` to produce the first tuple of the result. In the case where the right generating query is not among the very first candidate queries tested, such behavior could easily lead to subpar overall response time.

This section describes a filter based on probing queries. Probing queries are modifications of a given candidate query Q that aim to be processed much faster than the time needed for the first `getNext()` to generate the first tuple. As such, they might be capable of dismissing Q much quicker than the progressive technique alone. As we shall see, the idea of using probing queries bears some similarity to that of using progressive query processing.

Consider a candidate query Q = `SELECT` $c_1, c_2, \ldots, c_n$ `FROM` ... `WHERE` ... To formulate a probing query Q_{pr}, the algorithm selects a random tuple $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ from R_{out}. It then adds n additional conditions/constraints to the `WHERE` clause of Q in the form of $c_i = v_i$, for $i = 1, 2, \ldots, n$. Specifically, in the *leave-nothing-out* scheme, all of these conditions are present in Q_{pr}. Then, if Q is a generating query, it should generate R_{out} when applied on $\mathcal{D}$. Let Q_R = `SELECT` $*$ `FROM` R_{out} `WHERE` *conditions*, where *conditions* are the same $c_i = v_i$ conditions taken from Q_{pr}. Then, it should hold that $Q_{pr}(\mathcal{D}) = Q_R(R_{out})$ and if it does not, then Q cannot be a generating query and could be filtered away. Because probing queries are constrained versions of Q, they tend to be much faster than Q and serve as a good filter.

The leave-nothing-out scheme is similar in spirit to other probing queries used elsewhere, *e.g.,* [27, 38]. However, in its basic form, this technique has proven to be ineffective for `FastQRE`, especially when used in a combination with other filters. `FastQRE` thus uses an advanced probing methodology that is based on the ideas of (a) leveraging leave-one-out queries in addition to leave-nothing-out queries, and (b) using dynamic timeouts.

Leave-one-out Scheme. In the leave-one-out scheme, a randomly-chosen condition is dropped from a probing query among the aforementioned n conditions. Such queries tend to be more expensive to evaluate but much more effective at detecting wrong candidate queries. Hence, the approach issues a few leave-nothing-out and a few leave-one-out probing queries to perform the filtering.

However, simply using a mix of queries is not sufficient. One of the main challenges with probing queries is that, due to skew in data, their execution time often varies greatly depending on the chosen tuple $\mathbf{v}$. Some of these execution times could be substantial, even in the order of running $Q(\mathcal{D})$ test itself, defeating the purpose of this filtering step and even making the overall solution slower.

Timeout Mechanism. To address this problem, we could use a timeout mechanism, where a probing query is aborted if it runs for too long and then a different probing query is tried out. However, the main challenge is how to tune the timeout value dt. Notice, if dt is set too low, then all probing queries will time out, making the filter useless. If dt is set too high, this filter can become very expensive, even to the point where the approach performs better without the filter. What further complicates matters is that a good value of dt depends on both $\mathcal{D}$ and Q, making it hard to precompute and set dt once for all possible cases in advance. Thus, instead of fixing dt, we determine it *dynamically*: per $\mathcal{D}$ and Q, by using a timeout mechanism that adjusts dt based on query timeouts.

B ADDITIONAL EXPERIMENTS

Experiment 5 (FastQRE vs. Star: same hardware). The context for this experiment has been provided in Section 5, specifically in the part that describes the background of the `Star` system.

`Star` has been tested by its authors on a 128 GB Windows server for MySQL DBMS. In this experiment we test the original Star code on our setup which has 16 GB of RAM. `Star` is memory intensive and runs out of memory for most of the TPC-H queries. Thus, it has been able to reverse engineer only the 6 queries shown in Table 3.

Query	Old: 128GB	New: 16GB	Difference
TQ4	35.3 sec	50.3 sec	1.4× slower
TQ11	198.9 sec	150.6 sec	24% faster
TQ12	369.6 sec	290.8 sec	21% faster
TQ13	11.1 sec	53.1 sec	4.8× slower
TQ14	14.7 sec	45.9 sec	3.1× slower
TQ17	15.2 sec	30.8 sec	2.0× slower

Table 3: The results of Star on our 16GB machine.

Table 3 shows the old result for the 128 GB machine from [38], the new result on our 16 GB machine, and the difference between the two types of results. For example, for query TQ4 it shows that on the old machine it took 35.3 seconds for Star to reverse engineer it. For our 16GB machine this number is 50.3 seconds, which means the results have become 1.4× slower on our machine.

Query	Speedup
TQ4	26.5
TQ11	62.75
TQ12	181.8
TQ13	20.4
TQ14	23
TQ17	17.1

Table 4: Speedup of `FastQRE` over `Star`.

Table 4 shows the speedup of `FastQRE` over `Star`. It is computed as the processing time of `Star` divided by the processing time of `FastQRE`. We can see the speedup of 1-2 orders of magnitude, where the smallest speedup is 17.1 for query TQ17 and the largest speed up is 181.8 for query TQ12.

The experiment shows that `FastQRE` has a significant performance advantage over `Star`. It also shows that `FastQRE` is capable of reverse engineering more queries with a smaller RAM footprint.

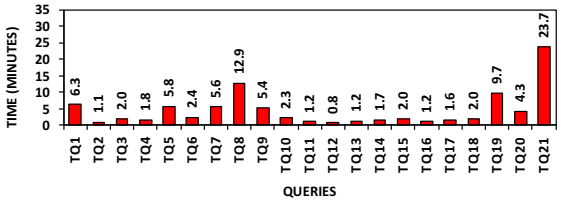


Figure 15: Execution time on `TPCH2`.

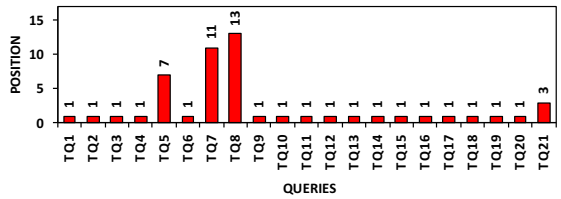


Figure 16: Quality of sequences on `TPCH2`.

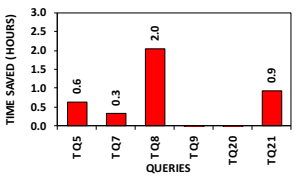


Figure 17: Time saved

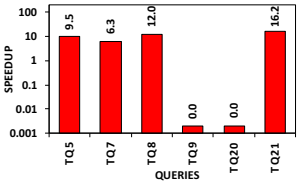


Figure 18: Speedup

Experiment 6 (Results on `TPCH2` Dataset). In this experiment we summarize the results of `FastQRE` on the original TPC-H dataset. We present experiments that are similar to the previous experiment on `TPCH1`. The changes in figures often reflect the differences between `TPCH2` and `TPCH1`. `TPCH2`'s values are less skewed than those of `TPCH1`, but `TPCH2` is about 9 times larger than `TPCH1`. Because of that, executing a single query is more expensive on `TPCH2`. For example, it takes 4 seconds to execute TQ8 on `TPCH1`, but it takes 2284 seconds (or 571 times more) to execute TQ8 on `TPCH2`.

Figure 15 studies the execution time of `FastQRE` on the `TPCH2`, excluding the time needed for the final $Q(\mathcal{D}) = R_{out}$ check. Compared to the corresponding results on `TPCH1`, the absolute values have increased given the increase in the size of data. However, in terms of its relative performance vs. the time needed to execute $Q(\mathcal{D})$, the results improve for `FastQRE` on `TPCH2`. Table 2 demonstrates that point: on `TPCH1` the final query check takes 34% whereas 66% is spent on the main logic. For `TPCH2` the final check is 64% and the main logic is only 36%. Thus, `FastQRE` fares well on `TPCH2`, especially given that it is 10 times larger than `TPCH1`.

Figure 16 is similar to Figure 12, but on `TPCH2` dataset instead of `TPCH1`. The differences between the two figures show that due to different value compositions in `TPCH1` and `TPCH2` the algorithm explores difference candidate queries for TQ5, TQ7, and TQ8.

Figures 17 and 18 study the absolute times saved and the speedup achieved by Algorithm 1 by using validation components. Compared to the result on `TPCH1`, the effect of the validation components becomes more pronounced for TQ5, TQ7, and TQ8, but becomes less for TQ21. For example, for TQ5 the speedup changes from 2.5 for `TPCH1` to 9.5 on `TPCH2`.