# Shedding Light on Opaque Application Queries

Kapil Khurana  Jayant R. Haritsa
Database Systems Lab, Indian Institute of Science, Bangalore 560012, India
{kapilkhurana,haritsa}@iisc.ac.in

## ABSTRACT

We investigate a new query reverse-engineering problem of unmasking SQL queries hidden within database applications. The diverse use-cases for this problem range from resurrecting legacy code to query rewriting. As a first step in addressing the unmasking challenge, we present UNMASQUE, an active-learning extraction algorithm that can expose a basal class of hidden warehouse queries. A special feature of our design is that the extraction is non-invasive wrt the application, examining only the results obtained from repeated executions on databases derived with a combination of data mutation and data generation techniques. Further, potent optimizations are incorporated to minimize the extraction overheads. A detailed evaluation over applications hosting hidden SQL queries, or their imperative versions, demonstrates that UNMASQUE correctly and efficiently extracts these queries.

## CCS CONCEPTS

• **Information systems** → **Query operators**; **Query intent**.

## KEYWORDS

SQL; Query Reverse Engineering; Black-Box Extraction; Imperative-Declarative Code Conversion; Active Learning

**ACM Reference Format:**
Kapil Khurana  Jayant R. Haritsa. 2021.Shedding Light on Opaque Application Queries. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457252

## 1 INTRODUCTION

Over the past decade, *query reverse-engineering* (QRE) has evinced considerable interest from both the database and programming language communities (e.g. [12, 14–16, 21, 24–26, 28, 30]). The generic problem tackled in this stream of work is the following: *Given a database instance $\mathcal{D}_I$ and a populated result $\mathcal{R}_I$, identify a candidate SQL query $Q_c$ such that $Q_c(\mathcal{D}_I) = \mathcal{R}_I$.* The motivation for QRE stems from a variety of use-cases, including: (i) reconstruction of lost queries; (ii) query formulation assistance for naive SQL users; (iii) enhancement of database usability through a slate of instance-equivalent candidate queries; and (iv) explanations for unexpectedly missing tuples in the result.


*SIGMOD '21, June 20–25, 2021, Virtual Event, China*

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457252

Impressive progress has been made on addressing the QRE problem, with the development of elegant tools such as TALOS [26], REGAL [25] and SCYTHE [28]. Notwithstanding, there are intrinsic challenges underlying the problem framework: First, the output query $Q_c$ is organically dependent on the specific ($\mathcal{D}_I$,$\mathcal{R}_I$) instance provided by the user, and can vary significantly based on this initial sample. Second, given the inherently exponential search space of alternatives, identifying and selecting among the candidates is not easily amenable to efficient processing.

## HQE Problem

In this paper, we consider a new variant of the QRE problem, wherein a *ground-truth* query is additionally available, but in a hidden form that is not easily accessible. For example, the original query may be explicitly hidden in a black-box application executable. Moreover, encryption or obfuscation may have been incorporated to further protect the application logic. Such "hidden-executable" situations could also arise in the context of legacy code, where the original source has been lost or misplaced over time, or when third-party proprietary tools are part of the workflow.

An alternative and more subtle scenario is that the application is visible but remains opaque because it is comprised of either (a) hard-to-comprehend SQL – such as those arising from machine-generated object-relational mappings, or (b) poorly documented *imperative code* that is not easily decipherable – which could occur when software is inherited from external developers.

Formally, we introduce the hidden-query extraction (HQE) variant of QRE as follows: *Given a black-box application $\mathcal{A}$ containing a hidden query $Q_H$ (in either SQL format or its imperative equivalent), and a database instance $\mathcal{D}_I$ on which $\mathcal{A}$ produces a populated result $\mathcal{R}_I$, unmask $Q_H$ to reveal the original query (in SQL format).* That is, in contrast to the speculative nature of standard QRE, we intend to find the precise $Q_H$ such that $\forall \boldsymbol{i}$, $Q_H(\mathcal{D}_i) = \mathcal{R}_i$.

The presence of the hidden ground-truth delivers a variety of benefits: (i) The output query now becomes independent of the initial ($\mathcal{D}_I$,$\mathcal{R}_I$) instance; (ii) Since the application can be invoked repeatedly on different databases, efficient and focused mechanisms can be designed to precisely identify $Q_H$; (iii) Even advanced SQL constructs – for instance, *pattern-based* (e.g. LIKE), *group-based* (e.g. HAVING), or *result-based* (e.g. LIMIT) – which fall outside the ambit of the traditional QRE framework, become amenable to capture; (iv) As a collateral utility, the revealed query can serve as a definitive starting input to database usability tools (e.g. TALOS [26]); (v) New use-cases related to database security and query rewriting become feasible, as highlighted in Section 2.

At first glance, it may appear that the existing QRE techniques could provide a *seed query* for HQE, followed by refinements to precisely identify the hidden query. However, as explained later in the paper, this is not practical for both conceptual and performance reasons. Therefore, we have approached the HQE problem from *first*

*principles*. Our experience is that the extraction is challenging due to factors such as: (a) acute dependencies between the various clauses of the hidden query, (b) possibility of schematic renaming, (c) result consolidation due to aggregation functions, and (d) presence of computed column functions.

```
Create Procedure tpch_HQ
with Encryption BEGIN
select l_orderkey, o_orderdate, o_shippriority,
       sum(l_extendedprice * (1 - l_discount))
       as revenue
from   customer, orders, lineitem
where  c_custkey = o_custkey
  and  l_orderkey = o_orderkey
  and  c_mktsegment = 'BUILDING'
  and  o_orderdate < '1995-03-15'
  and  l_shipdate > '1995-03-15'
group by l_orderkey, o_orderdate,
         o_shippriority
order by revenue desc, o_orderdate
limit 10 END
```

(a) **Hidden Query ($Q_H$)**

```
select  l_orderkey, o_orderdate, o_shippriority
        sum(l_extendedprice * (1 - l_discount))
        as revenue
from    customer, lineitem, orders
where   c_custkey = o_custkey
  and   l_orderkey = o_orderkey
  and   c_mktsegment = 'BUILDING'
  and   o_orderdate <= '1995-03-14'
  and   l_shipdate >= '1995-03-16'
group by  l_orderkey, o_shippriority,
          o_orderdate
order by  revenue desc, o_orderdate asc
limit 10
```

(b) **UNMASQUE Output Query ($Q_E$)**

**Figure 1: Hidden Query Extraction Example (TPC-H Q3)**

## UNMASQUE Extractor

We take a first step towards addressing the HQE problem here by presenting UNMASQUE[1], an algorithm that exposes the hidden query $Q_H$ through "active learning" – that is, by using the outputs of application executions on carefully crafted database instances. Specifically, UNMASQUE employs a judicious combination of *database mutation* and synthetic *database generation* to methodically expose the various query elements. The extraction is completely *non-invasive* wrt both the application software and the underlying database engine, facilitating portability.

At this time, UNMASQUE is capable of extracting a basal set of warehouse queries that feature the core SPJGHAOL[2] clauses – essentially, single-block equi-join queries with conjunctive predicates (the precise coverage is enumerated in Section 3). As an exemplar of a non-trivial query that falls in the ambit of its extraction scope, consider $Q_H$ in Figure 1(a) – this hidden query encrypts query **Q3** of the TPC-H benchmark in a stored procedure, outputting the top-ten unshipped orders with respect to revenue. Our extracted equivalent, $Q_E$, is shown in Figure 1(b) – we see that it clearly captures all *semantic* aspects of the original query, including the revenue column function. Only syntactic differences, such as a different order of the grouping columns, remain in the extraction.

A natural question here is what would a QRE tool, such as REGAL, output in this situation? Firstly, REGAL is intrinsically unable to handle the revenue function, apart from the ORDER BY and LIMIT constructs. Secondly, even after grossly simplifying the query by removing these constructs, as shown in Figure 2(a), REGAL produces the semantically dissonant output displayed in Figure 2(b) – while the tables and joins are detected correctly, significant discrepancies exist in the filters, grouping columns and aggregation functions. Moreover, as explained in [19], differences could arise in the tables and the joins as well, depending on the specific ($\mathcal{D}_I$,$\mathcal{R}_I$) instance that is supplied to the tool. Finally, even producing this limited outcome required considerable time and resources on a well-provisioned platform.

[1]Unified Non-invasive MAchine for Sql QUery Extraction
[2]SELECT, PROJECT, JOIN, GROUPBY, HAVING, AGG, ORDER, LIMIT

```
select  l_orderkey, o_orderdate,
        o_shippriority,
        sum(l_extendedprice)
from    customer, orders, lineitem
where   c_custkey = o_custkey
  and   l_orderkey = o_orderkey
  and   o_orderdate < '1995-03-15'
  and   l_shipdate > '1995-03-15'
group by l_orderkey, o_orderdate,
         o_shippriority
```

(a) **Simplified Query Input**

```
select  min(l_orderkey), sum(l_extendedprice),
        o_orderdate, min(o_shippriority)
from    customer, orders, lineitem
where   c_custkey = o_custkey
  and   l_orderkey = o_orderkey
  and   o_orderdate between '1994-11-19'
  and   '1995-03-10'
  and   l_shipdate between '1995-03-20'
  and   '1995-07-12'
group by o_orderdate
```

(b) **REGAL Output Query**

**Figure 2: REGAL-compliant QRE Example**

## Extraction Workflow

UNMASQUE operates according to the pipeline shown in Figure 3, where it methodically extracts the hidden query elements in a sequential manner. It starts with the FROM clause, continues on to the JOIN and FILTER predicates, follows up with the PROJECTION, GROUP BY and AGGREGATION columns, and concludes with the ORDER BY and LIMIT functions. (As explained in Section 7, a different pipeline structure is required to extract the HAVING predicate).

The initial elements (SPJ) are extracted using *database mutation* strategies, whereas the subsequent ones (GAOL) leverage *database generation* techniques. The final ASSEMBLER module combines and canonifies the elements of $Q_E$ into a standard output format, and performs a suite of correctness checks to verify the extraction.

## Extraction Efficiency

To ensure extraction efficiency, UNMASQUE incorporates a variety of optimizations. In particular, it addresses a conceptual problem of independent interest: *Given a database instance $\mathcal{D}_I$ on which $Q_H$ produces a populated result $\mathcal{R}_I$, identify the smallest subset $\mathcal{D}_{min}$ of $\mathcal{D}_I$ such that the result continues to be populated.*

Due to the hidden nature of $Q_H$, $\mathcal{D}_{min}$ cannot be obtained using standard provenance techniques (e.g. [17]). Therefore, we design alternative strategies based on a combination of sampling and recursive database partitioning to achieve the minimization objective.

The database minimization is applied immediately after the FROM clause is identified (Figure 3), ensuring that the subsequent SPJ extraction is carried out on *miniscule* databases containing just a handful of rows. Similarly, the synthetic databases created for GAOL extraction are also carefully designed to be very thinly populated. The net outcome is that the post-minimization processing becomes essentially *independent* of the original database size.

## Performance Evaluation

We have evaluated UNMASQUE's behavior on (a) complex SQL queries arising in synthetic (TPC-H, TPC-DS) and real (JOB-IMDB) environments, as well as (b) imperative code sourced from popular applications (Enki, Wilos, RUBiS). Our experiments, conducted on a vanilla PostgreSQL platform, indicate that the hidden queries are precisely identified in a timely manner. As a case in point, **Q3** was extracted on a 100 GB TPC-H instance within 10 minutes (native execution of this query took 5 minutes on the same platform).

A demo version of UNMASQUE was presented recently in [18], and a video of UNMASQUE in operation is available at [10].

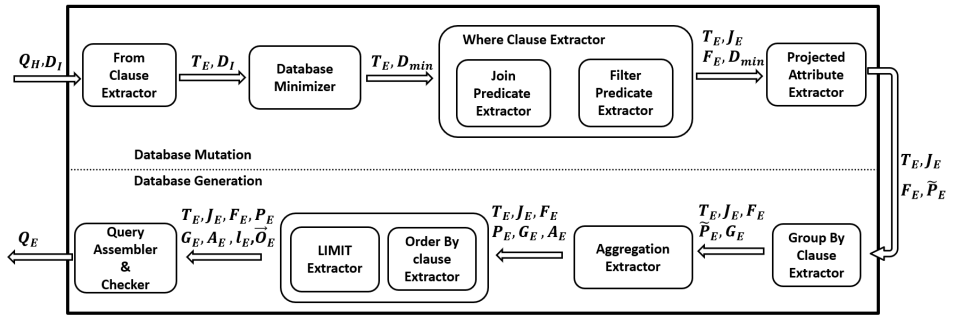


**Figure 3: UNMASQUE Architecture**

## Organization

In Section 2, a variety of extraction use-cases are outlined. A precise description of the HQE problem framework is provided in Section 3. The various components of the UNMASQUE pipeline are presented in Sections 4 and 5. The experimental framework and performance results are reported in Section 6. Extraction of the HAVING clause is discussed in Section 7, and a comparison to QRE approaches is reviewed in Section 8. Finally, our conclusions and future research avenues are summarized in Section 9.

## 2 HQE DEPLOYMENT SCENARIOS

We now discuss HQE deployment scenarios in the explicit and implicit opacity contexts. Our expectation is that the extraction process is invoked by the owner, or a privileged user, of the database, who possesses full access rights on the contents, including creation of a test silo for query extraction purposes.

### 2.1 Explicit Opacity

Here, only the executable object code of the application is available, and the goal is to determine the embedded data processing logic.

*Recovering Lost SQL.* It often happens, especially with legacy industrial applications, that with the passage of time, the original source code may be lost [4]. However, to understand the output, we may need to establish the logic connecting the database input to the observed result. Moreover, we may wish to extend or modify the existing application query, and create a new version.

If the SQL query is present as-is in the executable, it can be trivially extracted using standard string extraction tools (e.g. Strings [6]). However, if there has been post-processing, such as encryption or obfuscation, which are commonly resorted to for protecting application logic, this option is not feasible. For instance, the popular SQL Shield tool [5] offers encryption of stored SQL procedures on Microsoft SQL Server platforms.

Even with encrypted code, the query can be easily re-engineered from either the *execution plan* or the *query log* constructed by the database engine. However, this knowledge may also be inaccessible – as a case in point, SQL Shield blocks out both plan and log visibility for the encrypted query. Therefore, without direct access to the database internals, which is typically the situation with commercial database platforms or remotely-resident infrastructure (e.g. on the Cloud), the hidden query may not be easily extractable.

*Enhancing Database Security.* As highlighted in [20], SQL injection attacks often leverage query encoding techniques. For instance, an intruder can submit *"select * from passwords"* via embedded HEX-encoded strings. Further, encoded queries are also used to modify database security configurations to facilitate injection success.

Given such attack scenarios, UNMASQUE establishes a new approach to determining application objectives through query extraction, rather than directly resorting to heavy-weight and platform-specific techniques based on system audits and code forensics.

### 2.2 Implicit Opacity

Here, in addition to the executable, the application source is also available – however, the complex internal representation makes it effectively incomprehensible.

*Learning-based Query Rewriting.* The use of automated ORM tools is common today, often resulting in unintelligible SQL formulations [29]. Two problems arise from this artificial SQL complexity: (a) From a software engineering perspective, the code becomes hard to understand and maintain; (b) From a performance perspective, although query optimizers are supposed to convert these ORM queries to efficient equivalents, the plans generated are usually significantly slower. Modeling the rewriting as a HQE problem, however, makes it easy to construct a "lean" counterpart – this is because only databases and results are involved, and not the query itself. In this sense, HQE represents a viable *active-learning* technique for query rewriting, especially wrt "canned" workloads.

*Lightweight Code Conversion.* Due to either programming convenience or the lack of SQL expertise, software developers may choose to write *imperative code*. This may lead to serious execution inefficiencies since the database system's potent optimization abilities (e.g. indexes) are not utilized. The benefits of automated imperative-to-SQL conversion tools have been well recognized in recent times, resulting in their incorporation in mainstream database products (e.g. Froid [22] in SQL Server). However, these tools are typically host-language-specific (e.g. TSQL in Froid), and require support for special operators (e.g. Apply or Lateral) which may not be present on all engines, especially legacy ones. In contrast, UNMASQUE offers, over a restricted space of queries, a comparatively robust and generic approach to generating SQL from imperative code. This is because it is completely result-driven, making its usage both application and platform-independent.

## 3 PROBLEM FRAMEWORK

We assume an application executable object file is provided, containing either a single SQL query or imperative logic that is expressible in a single query. If there are multiple queries in the application, we require that each of them is invoked with a separate function call, and not batched together, reducing to the single query scenario.

Turning out attention to the database platform, we assume that it is freely accessible via its API, supporting all standard DML and DDL operations. There are no inherent restrictions on column data types, but for simplicity, we only consider here the common *numeric* (int, bigint, fixed-precision float), *character* (char, varchar, text) and *date* types. We assume the application owner provides a database instance that delivers a *populated* result – otherwise, we would have to resort to trial-and-error database generation to produce any result, an uncertain process with no convergence guarantees.

### 3.1 Extractable Query Class (EQC)

The QRE literature has primarily focused on constructing generic SPJGA queries featuring key-based equi-joins and conjunctive filter predicates. We share these basic structural restrictions, but our extraction scope is significantly enlarged to include HOL constructs, LIKE comparators, and multi-linear scalar functions. To achieve this extended coverage, we require additional mild assumptions: (i) Filter predicates feature only non-key columns and are of the type *column op value*. Further, for numeric columns, $op \in \{=, \leq, \geq, <, >, between\}$, whereas for textual columns, $op \in \{=, like\}$; (ii) The join graph is a subgraph of the schema graph (comprised of all valid PK-FK and FK-FK edges); (iii) All ordering columns appear in the projections; and (iv) The limit value is at least 3. We hereafter refer to this class of supported queries as *Extractable Query Class* (EQC).

Due to the difference in extraction frameworks, we initially present UNMASQUE for SPJGAOL queries, deferring the HAVING clause to Section 7. Accordingly, we qualify EQC as $EQC^{-H}$ (EQC without HAVING) in Sections 4 through 6. Further, for ease of exposition, we assume a slightly simplified framework in the subsequent description – for instance, that all keys are positive integer values – the extensions to the generic cases are provided in [19].

The notations used in our description of the extraction pipeline are summarized in Table 1, with the opaque application executable denoted by $\mathcal{E}$ to highlight its black-box nature.

### 3.2 Overview of Extraction Approach

To set up the extraction process, we first create a *silo* in the database with the same table schema as the original database. Subsequently, all referential integrity constraints are dropped from the silo tables, since the extraction process constructs alternative database scenarios that may be incompatible with the existing schema. We then create the following template representation for the to-be extracted query $Q_E$, as per the notation in Table 1 and assuming $EQC^{-H}$:

**Select** ( $P_E$, $A_E$ ) **From** $T_E$ **Where** $J_E \wedge F_E$
**Group By** $G_E$ **Order By** $\overrightarrow{O_E}$ **Limit** $l_E$;

The constituent elements of the template are sequentially identified, according to the extraction process shown in Figure 3.

The common theme across the SPJ extraction algorithms is that *atomic* changes are made to individual database columns and the effects on the result are observed – with regard to change in *cardinality* (Select, Join), or change in column *value* (Projection) – to establish presence in the respective clauses. On the other hand, for GAOL extraction, the common theme is that databases are created so as to assure *pre-determined* (albeit invisible) intermediate outcomes of the SPJ core of the query. Overall, this calibrated construction process supports precise establishment of associations between the input database columns and the output result columns.

The extraction details are explained in the following sections, using the introductory query, **TPC-H Q3**, as the running example. Due to space limitations, the proofs of correctness and the algorithmic complexity analysis are assigned to [19].

**Table 1: Notations**

| Symbol | Meaning | Symbol | Meaning (wrt query $Q_E$) |
|---|---|---|---|
| $\mathcal{A}$ | Application | $T_E$ | Set of tables in the query |
| $\mathcal{E}$ | Application Executable | $C_E$ | Set of columns in $T_E$ |
| $\mathcal{D}_I$ | Initial Database Instance | $JG_E$ | Join graph |
| $\mathcal{R}_I$ | Result of $\mathcal{E}$ on $\mathcal{D}_I$ | $J_E$ | Set of Join predicates |
| SG | Schema Graph of $\mathcal{D}_I$ | $F_E$ | Set of Filter predicates |
| $Q_H$ | Hidden Query | $G_E$ | Set of Group By columns |
| $Q_E$ | Extracted Query | $H_E$ | Set of Having predicates |
| $D_{min}$ | Reduced Database | $l_E$ | Limit value |
| $D^t$ | Database with at most $t$ rows in each table of $T_E$ | $P_E$ | Set of native Projections with mapped result columns |
| $D_{mut}$ | Mutated database | $A_E$ | Set of Aggregations with mapped result columns |
| $D_{gen}$ | Generated database | $\overrightarrow{O_E}$ | Sequence of Ordered result columns |

## 4 MUTATION PIPELINE

### 4.1 From Clause

To identify whether a base table $t$ is present in $Q_H$, the following elementary procedure is applied: First, $t$ is temporarily renamed to *temp*. Next, $\mathcal{E}$ is executed on this mutated schema and if an error is immediately thrown by the database engine, then $t$ is part of the query; otherwise, the ongoing execution is terminated after a short timeout period. Finally, *temp* is reverted to its original name $t$.

By iteratively checking all the database tables present in $\mathcal{D}_I$, $T_E$ is identified. For **Q3**, the outcome is $T_E$ = ***{customer, lineitem, orders}***. We hereafter use $\mathcal{D}_I$ to refer specifically to the part of the database relevant to $Q_E$ – i.e. the contents of $T_E$.

(Note: This approach works only for SQL applications – an alternative technique is presented for imperative code in [19].)

### 4.2 Database Minimization

For enterprise applications, it is likely that $\mathcal{D}_I$ is huge, and therefore repeatedly executing $\mathcal{E}$ during the extraction process may take an impractically long time. To tackle this issue, we initially invoke the MINIMIZER module to *minimize* the database as far as possible while maintaining a populated result. Specifically, we address the following **row-minimality** problem: *Given a database instance $\mathcal{D}_I$ and an executable $\mathcal{E}$ producing a populated result $\mathcal{R}_I$ on $\mathcal{D}_I$, derive a reduced database instance $D_{min}$ from $\mathcal{D}_I$ such that removing any row from any table leads to an empty or null result.* With this definition of $D_{min}$, the following strong observation can be made:

**Lemma 1:** For $EQC^{-H}$, there always exists a $D_{min}$ wherein each table in $T_E$ contains only a **single** row.

We hereafter refer to the single-row $D_{min}$ as $D^1$– an iterative reduction process to identify this database from $\mathcal{D}_I$ is explained next.

**Reducing $\mathcal{D}_I$ to $D^1$.** Pick a table $t$ from $T_E$ that contains more than one row, and divide it roughly into two halves. Run $\mathcal{E}$ on the first half, and if the result is populated, retain only this partition. Otherwise, retain only the second half, which must, by definition, have at least one result-generating row (due to Lemma 1). When eventually all the tables in $T_E$ have been reduced to a single row by this process, we have achieved $D^1$.

In principle, the tables in $T_E$ can be progressively halved in any order. However, after each halving, $\mathcal{E}$ is executed once to determine which half to retain, and we would like to minimize the time taken by these executions. Accordingly, we empirically evaluated various policies for which table to halve next (smallest table, largest table, random, etc.), and found that a policy of halving the *currently largest* table was usually the fastest way to reach the $D^1$ target.

**Sampling-based Preprocessing.** The efficiency of the above reduction strategy could be further improved by leveraging the *sampling* methods that are natively available in most database systems. Specifically, instead of executing MINIMIZER directly on $\mathcal{D}_I$, we first quickly reduce the initial database size by iteratively sampling from the large-sized tables, one-by-one in decreasing size order, until a populated result is obtained.

## 4.3 Equi-Join Predicates

To extract the key-based equi-join predicates $J_E$ of $Q_H$, we start with $SG$, the original schema graph of the database comprised of all semantically valid key-connecting edges. However, unlike the usual representation where the tables are the vertices, we use a finer granularity wherein the vertices are the *columns* in the tables. So, each edge $(u, v)$ denotes a join linkage between a pair of key attributes in the schema. (For composite keys, an edge is drawn from each key element to the corresponding destination column).

From $SG$, we create an (undirected) induced subgraph whose vertices are the key columns in $T_E$, and edges are the potential join linkages between these columns. Then, using the transitive property of inner equi-joins, this subgraph is converted through transitive closure into a collection of cliques. Finally, each clique is converted to a cycle graph, hereafter referred to as a cycle, by retaining any one of elementary $n$-length cycles ($n$ = number of nodes in the clique). Note that in our context, even the trivial elementary graph with $n = 2$ (a pair of nodes and an edge between them) is also considered to be a cycle. The complete collection of cycles is referred to as the *candidate join-graph*, or $CJG_E$.

Our motivation for the graph conversion step is that: (a) Checking presence of a connected component in the query is equivalent to verifying presence of the corresponding cycle, and (b) If a connected component is only partially present in the query, the simple cutting procedure outlined below can downsize it to smaller components.

We individually check the presence of each cycle from $CJG_E$ in the query, using the iterative procedure shown in Algorithm 1. Only those cycles which pass the test are retained in $JG_E$. The check is done with the following three steps:

(1) Using the CUT subroutine, a pair of edges, $e_1$ and $e_2$, is removed from a randomly chosen cycle $CYC$; this removal partitions $CYC$ into two connected components, and the new components are converted back into smaller cycles ($CYC_1$ and $CYC_2$) by reintroducing the relevant missing edge;
(2) Using the NEGATE procedure, negate (i.e. change the sign) all column values in $D^1$ corresponding to the vertices in $CYC_1$,
(3) Run $\mathcal{E}$ on this mutated database – if the result is empty, we conclude that at least one of the edges $e_1$ and $e_2$ is present in $JG_E$, and both $e_1$ and $e_2$ are returned to the parent cycle $CYC$; otherwise, $CYC$ is removed from $CJG_E$ and $CYC_1$ and $CYC_2$ are included as fresh candidates in $CJG_E$.

As a limiting case, if a cycle is reduced to a single edge, then the check is carried out by dropping the CUT step and using only NEGATE with one vertex of the edge.

In the above procedure, the motivation for removing a *pair* of edges is the following: For $JG_E$ to not contain a candidate cycle $CYC$, at least *two* edges of $CYC$ should be absent from the query – if only a single edge were to be removed, the cycle would still effectively remain by the *transitive* property of equi-joins. Note that the algorithm is guaranteed to terminate because, in each iteration, a cycle is either fully removed or partitioned into smaller cycles.

With regard to **Q3**, $CJG_E$ contains only two connected components – $(l\_orderkey, o\_orderkey)$ and $(o\_custkey, c\_custkey)$. Each component has a single edge that returns true when checked for presence by Algorithm 1. So, in this case, $JG_E \equiv CJG_E$. In the final step, each edge in $JG_E$ is converted into a predicate in $J_E$. Therefore, the equi-join predicates turn out to be:

$J_E$ = **{l_orderkey=o_orderkey, o_custkey=c_custkey}**.

**Algorithm 1:** Extracting Equi-Join Graph $JG_E$

```
CJG_E ← Candidate Cycles, JG_E ← φ
while There is at least one cycle in CJG_E do
    CYC ← Any candidate cycle from CJG_E
    if CYC contains a single edge (v1, v2) then
        D^1_mut ← Negate(D^1, {v1})
        If E(D^1_mut) = φ then JG_E ← JG_E ∪ CYC
        CJG_E ← CJG_E / CYC
    else
        foreach pair of edges (e1, e2) ∈ CYC do
            CYC1, CYC2 = Cut(CYC, e1, e2)
            D^1_mut ← Negate(D^1, CYC1)
            if E(D^1_mut) = φ then
                Add e1 and e2 back to CYC
            else
                CJG_E ← (CJG_E − CYC) ∪ CYC1 ∪ CYC2
                break    //Go to the start of while loop
            end
        end
        JG_E ← JG_E ∪ CYC;   CJG_E ← CJG_E / CYC
    end
end
```

## 4.4 Filter Predicates (non-key)

We iteratively check each (non-key) column in $C_E$, the set of columns in $T_E$, for its presence in a filter predicate (note that as per $EQC^{-H}$,

**Table 2: Filter Predicate Cases**

| Case | $\mid R_1 \mid = \phi$ | $\mid R_2 \mid = \phi$ | Predicate Type | Action Required |
|---|---|---|---|---|
| 1 | No | No | $i_{min} \leq A \leq i_{max}$ | No Predicate |
| 2 | Yes | No | $l \leq A \leq i_{max}$ | Find $l$ |
| 3 | No | Yes | $i_{min} \leq A \leq r$ | Find $r$ |
| 4 | Yes | Yes | $l \leq A \leq r$ | Find $l$ and $r$ |

each such attribute can appear in at most one filter predicate). The procedure for general numeric and textual predicates of the form *column op value* is explained below, with special cases such as NULL and boolean predicates discussed in [19].

*4.4.1* **Numeric Predicates.** For ease of presentation, we start by explaining the process for *integer* columns. Let $[i_{min}, i_{max}]$ be the value spread of column $A$'s integer domain, and assume a range predicate $l \leq A \leq r$, where $l$ and $r$ need to be identified. Note that all the comparison operators $(=, <, >, \leq, \geq, between)$ can be represented in this format (e.g. $A < 5 \equiv i_{min} \leq A \leq 4$).

To check for presence of a filter predicate on column $A$, we first create a mutated $D^1_{mut}$ instance by replacing the value of $A$ with $i_{min}$ in $D^1$, then run $\mathcal{E}$ and get the result – call it $R_1$. By applying the same process with $i_{max}$, we get a second result – call it $R_2$. Now, the existence of a filter predicate is determined based on matching, using the cardinalities of $R_1$ and $R_2$, one of the four disjoint cases shown in Table 2.

If the match is with Case 2, a binary-search is conducted over $(i_{min}, a]$ to identify the specific value of $l$, where $a$ is the value of column $A$ present in $D^1$. After this search completes, the associated predicate is added to $F_E$. Similarly, for Case 3, the search is over $[a, i_{max})$ to identify the value of $r$. Finally, Case 4 is a trivial combination of Cases 2 and 3, and handled in a similar manner.

Using a similar strategy, we can also extract *float* data types with *fixed precision*. Specifically, we first identify the integral bounds with the above procedure and then execute a second binary-search to identify the fractional bounds.

Finally, extraction of *date* predicates is identical to integers, the only change being the value limits and the difference unit (i.e. days).

*4.4.2* **Textual Predicates.** The extraction procedure for character columns is significantly more complex because: (a) strings can be of variable length, and (b) the filters may contain *wildcard* characters ('_' and '%'). To first check for the existence of a filter predicate on textual attribute $A$, we create two different $D^1_{mut}$ instances by replacing the value of $A$ with (a) an empty string, and (b) a single character string (say *"a"*), respectively. $\mathcal{E}$ is invoked on both these instances, and we conclude that a filter predicate is in operation iff the result is empty in at least one of the cases. To prove the *if* part, it is easy to see that if the result is empty in either of the cases, there must be some filter criteria on $A$. With regard to the *only if* part, the result will be populated for both cases in only one extreme scenario – A LIKE '%', which is equivalent to *no* filter on $A$.

After confirming existence of a filter predicate on $A$, the specific predicate is extracted in two steps. Before getting into the details, we define a term called *Minimal Qualifying String (MQS)* – given a character/string expression *str*, its MQS is the string obtained by removing all occurrences of '%' from *str*. For example, "UP_" is the MQS for *"%UP_%"*. With this notation, the first step is to identify MQS using the actual value of $A$ in $D^1$, denoted as the representative string, or *rep_str*. The idea here is to loop through all the characters of *rep_str* and determine whether it is present as an intrinsic character of MQS or invoked through wildcards ('_' or '%'). This distinction is achieved by replacing, in turn, each character of *rep_str* in $D^1$ with a different character, executing $\mathcal{E}$ on this mutated database, and checking if the result is empty – if yes, the replaced character is part of MQS; if no, this character was invoked through wildcards. The detailed algorithm is given in [19].

After obtaining the MQS, we need to find the locations (if any) in the string where '%' is to be placed to obtain the actual filter value. This is achieved with the following simple linear procedure: For each pair of consecutive characters in MQS, a random character that is different from both these characters is inserted between them. Then, we replace the current value in attribute $A$ with this new string. A populated result for $\mathcal{E}$ on this mutated database instance indicates the existence of '%' between the pair of characters. The inserted character is removed after each iteration and the initial MQS is used afresh for each successive pair of consecutive characters – this is done to ensure that the character length limit for $A$ is not exceeded. In the specific case of **Q3**, the *rep_str* value for c_mktsegment turns out to be the MQS itself, namely *'BUILDING'*.

Overall, the following numeric and textual filter predicates are extracted by the above procedures for query **Q3**:

$F_E$ = **{ o_orderdate ≤ date '1995-03-14' ,**
**l_shipdate ≥ date '1995-03-16' ,**
**c_mktsegment = 'BUILDING' }**

From hereon, we will refer to attribute values that satisfy the corresponding filter and join predicates in the query as **s-values**. For attributes without filters, including key attributes on which filters are not permitted in $EQC^{-H}$, the s-values can be sourced from their entire domains. Our subsequent extractions are carried out only on databases populated with s-values.

## 4.5 Projection Columns

Identification of projections is tricky since they appear in a variety of different forms – native database columns, renamed columns, aggregated columns, and computed columns. To have a unified extraction procedure, we begin by treating each result column as an (unknown) constrained scalar function of one or more database columns. We explain here the procedure for identifying this function, assuming *linear* dependence on the column variables and at most *two* columns featuring in the function – the extension to an arbitrary number of columns is discussed in [19].

Let $O$ denote the visible output column, and $A, B$ the (unknown) database columns that may affect $O$. Given our assumption of linearity, the function connecting $A$ and $B$ to $O$ can be expressed with the following equation structure:

$$aA + bB + cAB + d = O \tag{1}$$

where $a, b, c, d$ are constant coefficients. Note that there may be an additional aggregation on the function (as in the revenue function of **Q3**), but its effect is nullified due to $D^1$ being a *single-row* database. With this framework, the extraction process proceeds, as explained below, in two steps: (i) Dependency List Identification, which establishes the identities of $A$ and $B$, followed by (ii) Function Identification, which computes the values of $a, b, c, d$.

*4.5.1* **Dependency List Identification.** In this step, for each output column $O$, the set of database columns which affect its value is discovered via iterative column exploration and database mutation. Specifically, the s-value of each database column in $D^1$ is mutated in turn to evaluate its impact on the value of $O$ – if there is a change, then $O$ is dependent on this column. If not, the exercise is repeated a *second* time with new s-values to ensure that the lack of change is not due to a *coincidental* original value of one column preventing impact of the other – for example, if $A = \frac{-b}{c}$ in $D^1$, then $B$ has no impact on $O$, irrespective of its value.

Using the above procedure on **Q3**, the following dependency lists are obtained for the various output columns: *l_orderkey: [l_orderkey], o_orderdate: [l_orderkey], o_shippriority: [o_shippriority],* and *revenue: [l_extendedprice, l_discount].*

*4.5.2* **Function Identification.** With reference to Equation 1, at this stage we are aware of the identities of $A$ and/or $B$ for each of the output columns, and what remains is to obtain the coefficient values $a, b, c, d$. This can be easily solved by creating 4 different mutated $D^1_{mut}$ instances such that the resultant equations are linearly independent. To do so, we randomly mutate the s-values of $A$ and $B$, each time checking whether the new vector $[A, B, AB, 1]$ is linearly independent from the vectors generated so far, and stop when four such vectors have been found.

For **Q3**, applying the above random procedure on the revenue output column, which depends on $A = l\_extendedprice$ and $B = l\_discount$, we happen to obtain the following system of equations:

$$1.a + 2.b + 2.c + d = -1 \qquad 2.a + 1.b + 2.c + d = 0$$
$$2.a + 3.b + 6.c + d = -4 \qquad 1.a + 4.b + 4.c + d = -3$$

Solving this system results in the coefficient values: $a = 1, b = 0, c = -1, d = 0$, producing the function seen in the query. For the remaining output columns, which are all dependent on only a single database column, we get a function of the form $aA + d$ with $a = 1, d = 0$ – i.e. a native database column.

Thus, we obtain for query **Q3**, the following projection columns:

$\widetilde{P_E}$ = **{l_orderkey: l_orderkey, o_orderdate: o_orderdate, o_shippriority: _shippriority, revenue: l_extendedprice * (1 - l_discount) }**.

The reason the above set is shown as $\widetilde{P_E}$, and not $P_E$, is that some of these projections are subsequently refined as aggregations ($A_E$) in the Generation Pipeline – for instance, revenue becomes a *sum*. We reiterate that aggregations could be ignored thus far because our extraction techniques operated on *single-row* databases, making all aggregation functions identical with regard to their values.

# 5 GENERATION PIPELINE

## 5.1 Group By Columns

Our generic approach is that for each attribute $t.A$ in $C_E$ (the set of columns in $T_E$), we create a tiny synthetic database instance $D_{gen}$ and analyze $\mathcal{E}(D_{gen})$ for the existence of $t.A$ in $G_E$, the columns in the GROUP BY clause. However, this check is skipped for columns with equality filter predicates (as determined in the Mutation Pipeline) since their presence in $G_E$ is superfluous. Assume for the moment that we have constructed a $D_{gen}$ such that the (invisible) *intermediate* result produced by the SPJ core of $Q_H$

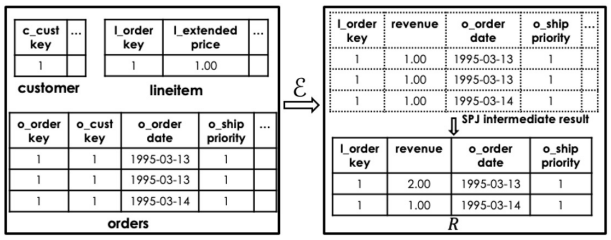


**Figure 4: $D_{gen}$ for Grouping on** o_orderdate **(Q3)**

contains **3** rows satisfying the following condition: Column $t.A$ has a common value in exactly two rows, while all other columns have the same value in all three rows. Now, if the *final* query result contains **2** rows, it means that this grouping is only due to the two different values in $t.A$, making it part of $G_E$. This approach to intermediate result generation is similar to techniques in [23, 27] for identifying query mutants.

**Creating $D_{gen}$.** We now explain how to create the desired $D_{gen}$ for checking $G_E$ membership. The procedure is specific to the presence or absence of $t.A$ in $JG_E$, the query join graph identified in the Mutation Pipeline, leading to the two cases described below. In the following description, assigning $(p, q, r, ...)$ to $t.A$ means assigning s-value $p$ in the first row, $q$ in the second, $r$ in the third and so on.

**Case 1: $t.A \notin JG_E$.** Here, we generate 3 rows for table $t$ and only 1 row in each of the other tables in $T_E$. For column $t.A$, any two distinct s-values $p$ and $q$ are first chosen, and then $(p, p, q)$ is assigned to $t.A$. Next, for all columns $t.B \in JG_E$, a fixed s-value of $r = 1$ (since keys do not feature in filter predicates) is assigned in all 3 rows. Finally, in all remaining columns of $t$, a random s-value $r$ is selected and assigned to all 3 rows.

Then, for the remaining 1-row tables $t'$ in $T_E$, a fixed s-value of $r = 1$ is assigned to all columns $t.B \in JG_E$, while random s-values are assigned to the remaining columns.

An example $D_{gen}$ for checking presence of o_orderdate in the $G_E$ of **Q3** is shown in Figure 4. Here, the ORDERS table features 3 rows with $p$ = '1995-03-13' and $q$ = '1995-03-14', while the LINEITEM and CUSTOMER tables have a single row apiece.

**Case 2: $t.A \in JG_E$.** Here, we generate 3 rows for table $t$, 2 rows for all tables $t'$ having a column $t'.B$ with a path between $t.A$ and $t'.B$ in $JG_E$, and only 1 row for the remaining tables in $T_E$. The assignment of values in the tables is similar to Case 1 with the following modifications: (i) In $t.A$, $p$ and $q$ are assigned fixed s-values of 1 and 2, respectively, in the 3 rows; (ii) Each column $t'.B$ is assigned fixed s-values $(1, 2)$ in its two rows, and the remaining columns in its table are assigned duplicated random s-values.

An example $D_{gen}$ for checking the presence of l_orderkey in $G_E$ is shown in Figure 5. Here, there are 3 rows for LINEITEM, 2 rows for ORDERS and 1 row for CUSTOMER.

It is straightforward to see by inspection that, with a restriction to key-based equi-joins, the above data generation procedure ensures the desired intermediate SPJ result. Namely, it contains exactly 3 rows with all columns having identical values in these rows, except the attribute under test which has *two* values in them.

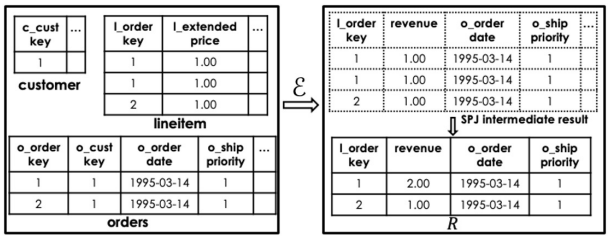


**Figure 5: $D_{gen}$ for Grouping on** l_orderkey **(Q3)**

It is possible that after all attributes have been processed in the above manner, $G_E$ remains empty. In this case, we create another $D_{gen}$ with each table having *two rows*, assigning fixed s-values $(1, 2)$ to each column in $JG_E$, a matching s-value to each column with an equality filter predicate, and any two different s-values to all other columns. Then, $\mathcal{E}$ is run on this $D_{gen}$, and if the result contains just 1 row, we conclude the query features an *ungrouped* aggregation.

Overall, for **Q3**, the above procedure results in:

$G_E$ = **{l_orderkey, o_shippriority, o_orderdate}**.

## 5.2 Aggregation Functions

We now explain the procedure for identifying the basic SQL aggregations – *min(), max(), count(), sum(), avg()*. Due to space limitations, only numeric attributes are discussed here, but similar methods can be used for textual and other attributes as well. Further, for ease of presentation, we assume that there is no DISTINCT aggregation, deferring this case to [19].

The aggregation identification proceeds as follows: Let $O = agg(f_o(A_1, ..., A_n))$, where *agg* corresponds to the aggregation, and $f_o(A_1, ..., A_n)$ to the projection function identified in Section 4.5 on database columns $A_1, ..., A_n$. Our goal now is to create a database $D_{gen}$ such that the *final* result row-cardinality is **1**, and each of the five possible aggregation functions on $f_o$ results in a *unique value*, thereby facilitating correct identification of the specific aggregation. We call this the desired "target result".

To distinguish between *min()* and *max()*, at least two different values are required in the input database columns. Further, to ensure unique values for the various output aggregations, we do as follows: Consider a pair of input s-value vector arguments $(s_1, .., s_i, .., s_n)$ and $(s_1, .., s'_i, .., s_n)$ such that $f_o(s_1, .., s_i, .., s_n) = o_1$ and $f_o(s_1, .., s'_i, .., s_n) = o_2$, with $o_1 \neq 0$, $o_1 \neq o_2$. Now assume we have generated a database $D_{gen}$ such that there are $k+1$ rows in the (invisible) *intermediate* result produced by the SPJ core of the query, with value $f_o = o_1$ in $k$ rows and $f_o = o_2$ in the remaining row. We ensure that each aggregation is guaranteed to produce a different value, by assigning $k$ to be the smallest positive integer that does *not* coincide with any of the following "forbidden" values (derived by computing pairwise equivalences among the five aggregation functions, as detailed in [19]):

$$\left\{0, o_1 - 1, o_2 - 1, 1 - \frac{o_2}{o_1}, \frac{1 - o_2}{o_1 - 1}, \frac{(o_1 - 2) \pm \sqrt{(o_1 - 2)^2 + 4(o_2 - 1)}}{2}\right\} \tag{2}$$

Finally, all $G_E$ grouping attributes are assigned common values in all the rows.

The above procedure ensures that the outcome of $\mathcal{E}$ on $D_{gen}$ is the target result because: (i) The result row-cardinality is 1 since there is a common set of values for the $G_E$ attributes, and (ii) The constraints on $k$ ensure unique aggregated outputs for all possible aggregations in $O$.

Note that as a special case, if $f_o$ is a constant function, then by definition $o_1 = o_2 = c$. More subtly, this is also the case when $f_o$ is a function of only the columns in $G_E$, because each unique combination of $G_E$ values forms a different group. In this scenario, the forbidden-value constraint reduces to $k \notin \{0, c - 1\}$. Further, the *min(), max(), avg()* functions are all equivalent due to the group-wise computation of the aggregation, and therefore *any* of these functions can be taken as the final choice – in our canonical format, we have chosen to use *min()*.

**Generating $D_{gen}$.** First, we choose the $i^{th}$ function argument $A_i$ to be a column that is not in $G_E$. If such an $A_i$ is not available, then as mentioned above, $s_i = s'_i$ and any argument column can be chosen as $A_i$. Next, we pick any two of the arguments that were used to identify dependency lists for $f_o$ (Section 4.5) if they satisfy the above-mentioned output condition, or generate a new set of compliant arguments. Subsequently, $k$ is chosen as the least positive integer satisfying Equation 2. Finally, the data generation process to obtain the desired intermediate result is similar to the $D_{gen}$ generation of GROUP BY (Section 5.1), with the following changes:

- $k + 1$ rows are generated for table $t$ where $A_i \in t$, with $t.A_i$ assigned value $s_i$ in $k$ rows and $s'_i$ in the remaining row.
- The other argument database columns, $A_j$ s.t. $j \neq i$, are assigned corresponding $s_j$ values in all the rows.
- With respect to Case 2 ($t.A_i \in JG_E$), all assignments of fixed values 1, 2 are replaced with values $s_i, s'_i$.

A sample $D_{gen}$ to check for aggregation on l_extendedprice * (1 - l_discount) is shown in Figure 6. Here, $k = 1$ and (l_extendedprice, l_discount) is set to $< (3, 0), (4, 0) >$. We run $\mathcal{E}$ on this $D_{gen}$ and the aggregation is identified by matching $O$'s value with the corresponding unique values for the five aggregations – in this case, it turns out to be *sum()*.

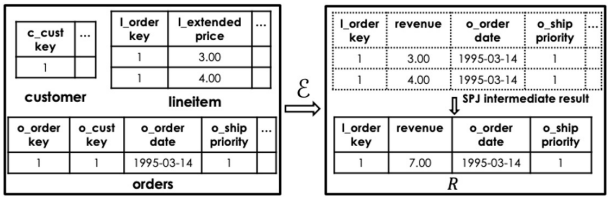


**Figure 6: $D_{gen}$ for Aggregation on revenue UDF (Q3)**

In the last step, the entries corresponding to the aggregated columns are removed from $\widetilde{P_E}$ and inserted in $A_E$, along with their associated functions. Further, if an unmapped output column is present in $\widetilde{P_E}$, it is removed and $count(*)$ is added to $A_E$. Whatever remains in $\widetilde{P_E}$ constitutes the native (i.e. unaggregated) $P_E$.

With the above procedure, we finally obtain for **Q3**:

$A_E$ = **{revenue: sum(l_extendedprice * (1 – l_discount))}**

$P_E$ = **{l_orderkey:l_orderkey, o_orderdate:o_orderdate, o_shippriority:o_shippriority}**

## 5.3 Order By

We now move on to identifying the sequence of columns present in $\overrightarrow{O_E}$. A basic difficulty here is that the result of a query can be in a particular order either due to: (i) an explicit ORDER BY clause in the query or (ii) a particular plan choice (e.g. Index-based access or Sort-Merge join). Given our black-box environment, it is *fundamentally infeasible* to differentiate the two cases. However, even if extraneous orderings appear in $Q_E$ due to the plan, the query semantics will not be altered, and so we allow them to remain as such.

In the following presentation, we expect that each database column occurs in the dependency list of at most one output column. Further, for simplicity, we assume that $count() \notin A_E$ – the procedure to handle this special case is given in [19].

*5.3.1 Order Extraction.* We start with a candidate list comprised of the output columns in $P_E \cup A_E$. From this list, the columns in $\overrightarrow{O_E}$ are extracted sequentially, starting from the leftmost index. The process stops when either (i) all candidates or functionally-independent attributes of $G_E$ have been included in $\overrightarrow{O_E}$, or (ii) no sort order can be identified for the current index position.

To check for the existence of an output column $O$ in $\overrightarrow{O_E}$, we create a pair of **2-row** database instances – $D^2_{same}$ and $D^2_{rev}$. In the former, the sort-order of $O$ is the *same* as that of all the other output columns, whereas in the latter, the sort-order of $O$ alone is *reversed* with respect to the other output columns. An example instance of this database pair is shown for the revenue function in Figure 7.

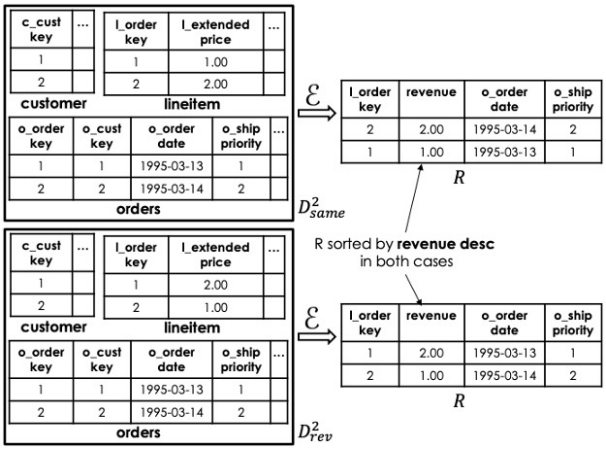


**Figure 7: $D^2_{same}$ and $D^2_{rev}$ for Ordering on** revenue **(Q3)**

**Creating $D^2_{same}$.** We use the following procedure to create $D^2_{same}$: First, the output columns are partitioned into three sets: (i) $S_1$, the set of output columns already present in $\overrightarrow{O_E}$ (initially, $S_1 = \phi$); (ii) $S_2$, a singleton set containing the output column currently under analysis; and (iii) $S_3$, the set of all remaining output columns. Let $f_o$ denote the function for output column $O$ (as per Section 4.5). For each $O \in S_1$, we select a common s-value vector $\overrightarrow{V_0} = s_1, s_2, ..., s_n$ to populate the argument columns present in $f_o$. Whereas, for each $O \in S_2 \cup S_3$, we select a *pair* of s-value vectors $\overrightarrow{V_1}$ and $\overrightarrow{V_2}$ such that each vector returns a different value in the output column.

The data generation for the tables is as follows:

- The argument columns for output column $O \in S_1$ are assigned $\overrightarrow{V_0}$ in both the rows.
- The argument columns for output column $O \in S_2 \cup S_3$ are assigned $\overrightarrow{V_1}$ and $\overrightarrow{V_2}$ in the two rows such that the output columns in $S_2 \cup S_3$ are all sorted in the same order.
- For the remaining columns with equality filter predicates, the single qualifying s-value is assigned in both the rows.
- For all other columns, a pair of random s-values $p$ and $q$ is assigned to the two rows with $p < q$. As always, consistency across connected key attributes is maintained by assigning the same $p, q$ pair to the matching attributes.

The procedure for creating $D^2_{rev}$ is the same as that for $D^2_{same}$ except that the argument attributes corresponding to the output column in $S_2$ are assigned values in the *reverse* order (i.e. $\overrightarrow{V_2}$, $\overrightarrow{V_1}$).

This database construction mechanism ensures that the two input rows eventually form individual output groups. Therefore, all aggregated columns can be effectively treated as projections (except $count()$, which requires a different mechanism, explained in [19]). After generating $D^2_{same}$ and $D^2_{rev}$, we run $\mathcal{E}$ on both instances and analyze the results, $R_{same}$ and $R_{rev}$. If the values in $O$ are sorted in the same order in both $R_{same}$ and $R_{rev}$, $O$ along with its associated direction (asc or desc) is added to $\overrightarrow{O_E}$ at position $i$. The sets $S_1, S_2$ and $S_3$ are then recalculated for the next iteration.

With the above procedure, we finally obtain for **Q3**:

$\overrightarrow{O_E}$ = **{revenue desc, o_orderdate asc}**

## 5.4 Limit

For an SPJGAOL query, extracting $l_E$ requires generating a database instance such that $\mathcal{E}$ produces more than $l_E$ rows in the result, subject to an upper bound imposed by the GROUP BY clause.

The maximum number of different values a column can legitimately take is a function of multiple parameters – data type, filter predicates, database engine, hardware platform, etc. Let $n_1, n_2, n_3, ..$ be the number of different values, after applying domain and filter restrictions, that the functionally-independent attributes $A_1, A_2, A_3, ..$ in $G_E$ can respectively take. This means that there can be a maximum of $n_1 * n_2 * n_3 * ... = l_{max}$ groups in the result. Thus, $l_E$ values up to $l_{max}$ can be extracted with this approach.

To extract $l_E$, UNMASQUE iteratively generates database instances such that the result-cardinality follows a *geometric progression* starting with $a$ rows and having common ratio $r$. We set $a = max(4, |\ R_I\ |)$ to accommodate $G_E$'s extraction requirement of 3 row results. And $r$ is set to balance the number of iterations with the setup cost for each iteration (in our experiments, $r = 10$).

**Generating $D_{gen}$ for desired $R$ cardinality.** To get $n$ result rows prior to the limit kicking in, we generate a database instance with each table having $n$ rows such that the functionally-independent attributes in $G_E$ have a unique permutation of values in each row. Specifically, all attributes appearing in $JG_E$ are assigned values $(1, 2, 3, ..., n)$, while other attributes are assigned random s-values. After applying $\mathcal{E}$ on this database, if the result contains $m$ rows with $m < n$, we conclude that a LIMIT of $m$ is operational.

Applying this procedure for **Q3**, we obtain $l_E$ = **10**.

## 5.5 Query Extraction Checker

In the final module, we conduct a suite of automated tests to verify the extraction correctness. First, several randomized large databases are created on which both the application and the extracted query are run. The results are compared, and a non-zero difference indicates an error. Further, physical ordering equivalence is verified by computing position-dependent checksums on the results.

Second, we leverage the XData grading tool [23], which verifies equivalence of student queries wrt to a model solution by constructing a suite of small test databases that are capable of detecting even subtle semantic differences in the respective query constructions. In our context, the extracted query is mapped to the model solution, and the hidden application to the student version.

# 6 EXPERIMENTS

We now move on to empirically evaluating UNMASQUE's efficacy and efficiency – our experiments are carried out on a vanilla PostgreSQL 11 database platform (Intel Xeon 2.3 GHz CPU, 32GB RAM, 3TB Disk, Ubuntu Linux) with default primary-key indices.

## 6.1 Comparison with QRE techniques

To begin with, we compare UNMASQUE against QRE techniques – specifically, the state-of-the-art REGAL tool [24, 25, 30]. Firstly, as already mentioned in the Introduction, there are considerable semantic differences between the query outputs of these two techniques. Secondly, we found that on large databases, REGAL either took several hours to complete or in some cases prematurely stopped due to running out of memory. Moreover, even on a small 5 GB database size, our extraction was significantly faster, often by an order of magnitude. This is quantified in Figure 8, which shows the performance of UNMASQUE and REGAL on 11 queries – RQ1 through RQ11 – that are compliant with the query templates used in [24]. As a case in point, REGAL took close to 800 seconds for RQ1, whereas UNMASQUE completed the extraction in only 25 seconds. Moreover, despite the reduced database size, REGAL did not complete a few queries, denoted by DNC in the figure.

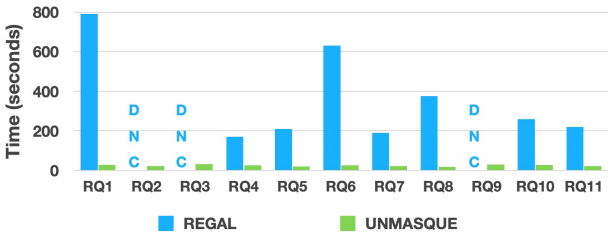


**Figure 8: Comparison with QRE (5 GB)**

We have also conducted a similar comparison with the TALOS tool on datasets from the UCI archive [9]. The performance results, which are detailed in [19], are consistent with those obtained for REGAL. Due to these large qualitative and quantitative differences wrt QRE systems, we present only the UNMASQUE profile in the remaining experiments of this section.

## 6.2 Hidden SQL Queries

Our primary extraction experiments were conducted on a basal suite of complex SPJGAOL queries, which are all $EQC^{-H}$-compliant and derived from the following popular benchmarks: (a) **TPC-H** [8] (12 queries), (b) **TPC-DS** [7] (7 queries), and (c) **JOB** [2] (11 queries), constructed on the real-world IMDB movie dataset. Each query was passed through a Cpp program that embedded the query in a separate executable, which formed the input to UNMASQUE. We then compared, both manually and through the automated tests discussed in 5.5, the extracted output with the original query and verified semantic equivalence.

The extraction times are analyzed below – only TPC-H and JOB are presented here, while TPC-DS results are reported in [19].

*TPC-H.* These hidden queries are similar in complexity to the **Q3** running example, and are listed in [19]. For convenience, we hereafter refer to them as *Qx*, where *x* is their associated TPC-H query identifier. The total end-to-end time taken to extract each of the 12 queries on a 100 GB initial instance (with a populated result) is shown in Figure 9. In addition, the breakup of the primary pipeline contributors to the total time is also shown in the figure.

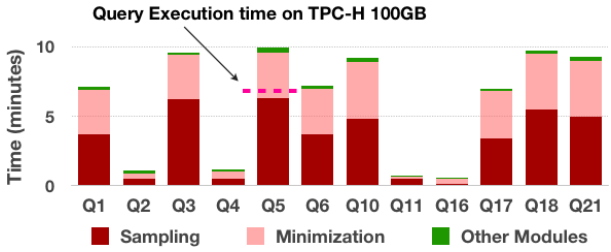


**Figure 9: Hidden Query Extraction Time (TPC-H 100GB)**

We first observe that the extraction times are practical for offline analysis environments, with all extractions being completed within **10 minutes**. Secondly, there is a wide variation in the extraction times, with some being completed in under 1 minute (e.g. Q16). The reason is the presence or absence of the lineitem table in the query – this table is enormous in size (around 0.6 billion rows), occupying about 80% of the database footprint, and therefore inherently incurring heavy processing costs.

Drilling down into the performance profile, we find that the MINIMIZER module of the pipeline takes up the lion's share of the extraction time, spread roughly evenly across the sampling (maroon color) and iterative partitioning (pink color) efforts. The remaining modules (green color) collectively complete within just a few seconds. For instance, with Q5 which consumed the maximum of 10 minutes, the MINIMIZER expended 98 % of this time, and only a paltry 2% was taken by all other modules combined.

The extreme skew across the modules is because the MINIMIZER operates on the original large database, whereas the other modules work with miniscule mutations or synthetic constructions. We also counted the number of times $\mathcal{E}$ was invoked to completion during the extraction – it was typically a *few hundred* times – but in spite of these numerous invocations, the overheads turned out to be negligible due to the tiny database sizes involved in the executions.

An alternative testimonial to UNMASQUE's efficiency is obtained by comparing the total extraction times with their corresponding query response times. For all the queries in our workload, this ratio was less than **1.5**. As a case in point, executing Q5 on the 100GB database took 6.5 minutes, shown by the red dashed line in Figure 9, in the same ballpark as the 10 minute extraction time.

*Join Order Benchmark (JOB).* The 11 queries we evaluated on the JOB benchmark, which is based on the IMDB database (5 GB), are characterized by extremely rich join-graphs, with $\geq 7$ joins in each query – in fact, query **Q24b** has as many as **12 joins**! Despite this complexity, they were all correctly extracted in less than **3 minutes** apiece, as shown in Figure 10. Again, the initial database size reduction takes the lion's share of the extraction, with the remaining modules collectively completing in less than a minute.

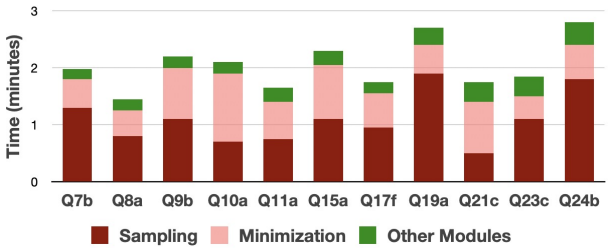


**Figure 10: Hidden Query Extraction Time (JOB)**

**Database Scaling.** To assess UNMASQUE's scaling ability, we also conducted the same set of extraction experiments on a suite of different-sized TPC-H instances, starting from 200 GB and going up to 1 TB in increments of 200 GB. A sample result corresponding to Q5, the most difficult extraction, is shown in Figure 11. We observe here that UNMASQUE delivers a quasi-linear behavior with a gentle slope, completing 1 TB in less than 25 minutes. In contrast, the native execution of Q5 has a much sharper slope, taking around 72 minutes on 1 TB, almost 3 times the query extraction time.

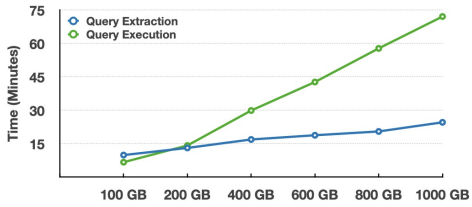


**Figure 11: Extraction Scaling Profile Q5 (TPC-H)**

**Schema Scaling.** Enterprise warehouses often contain hundreds of tables in their schema. Therefore, a legitimate concern is that the From clause extraction may be impractically slow in such environments. To assess this issue, we added 1000 tables to the TPC-H schema, and evaluated the $T_E$ identification overheads for a query having 12 tables – with an execution timeout setting of 100 milliseconds, the process completed in less than **10 seconds**.

## 6.3 Hidden Imperative Code

Our second set of experiments evaluated the following large-scale applications hosting imperative code: (a) **Enki** [1], a blogging application built with Ruby on Rails, (b) **Wilos** [11], a process orchestration software based on the Hibernate ORM, and (c) **RUBiS** [3], an auction site benchmark – due to space limitations, we only cover Enki and Wilos here, with RUBiS available in [19].

*Enki.* This application has commands to navigate pages, posts and comments. Since native data is not publicly available, we created a synthetic 10 MB database that provided populated results for all these commands. We found 14 out of 17 Enki commands to be compatible with our extraction scope, and verified that the correct SQL query was output for each of these imperative commands. As a sample instance, a snippet of *"find_recent"* function, invoked by *"get latest posts by tag"* command, is outlined in Figure 12(a). The corresponding UNMASQUE output is shown in Figure 12(b), and was produced in just 3 seconds. The remaining commands were all also extracted in just a few seconds, and their timings are listed in [19] along with the complexity of the extracted queries.

```
def find_recent(options = {})
  ...
  if_tags = options[:include] == :tags
  order = 'published_at DESC'
  cond = ['published_at < ?', Time.zone.now]
  limit = options[:limit] ||= DEFAULT_LIMIT
  result = Post.tagged_with(tag)
  result = result.where(cond)
  result = result.includes(:tags) if if_tags
...
```

(a) **Imperative Code (snippet)**

```
select min(posts.title), min(posts.body),  count(*),
       max(posts.published_at), min(tags.name)
from   posts, comments, taggings, tags
where  posts.id = comments.post_id
   and taggings.tag_id = tags.id
   and posts.id = taggings.taggable_id
   and taggable_type = 'Post'
   and (posts.published_at < cur_timestamp)
group by comments.post_id
order by max(posts.published_at) desc
limit 15
```

(b) **Extracted Equivalent SQL**

**Figure 12: Imperative to SQL Conversion – Enki**

*Wilos.* This Java-based application has been previously used to showcase the potential of imperative-to-SQL conversion tools such as DBridge [13] and QBS [14]. There are 33 code snippets listed in [14], with each snippet consisting of a function call internalizing a single query – of these, 22 were found to be compatible with our extraction scope. A synthetic database of size 10 MB was created, and the results of the in-scope functions on this database were serialized into database tables. Further, the table and the row corresponding to the function's input object were taken as constant since Wilos uses only this specific row as input from the source table.

We verified that UNMASQUE was able to correctly identify the equivalent queries for all 22 functions, accomplishing these extractions within a few seconds. As quantitative evidence, Table 3 shows the SQL clauses appearing in the 9 most complex functions, along with the associated extraction timings.

**Table 3: Imperative to SQL Conversion – Wilos**

| File (Function Line No.) | Extracted SQL Complexity | Time (sec) |
|---|---|---|
| ActivityService (347) | Project, Join, Group By, Order By | 1.7 |
| Guidance Service (168) | Project, Join, Group By | 2.1 |
| ProjectService (297) | Filter, Project, Join, Group By | 2.4 |
| ConcreteActivityService (133) | Project, Join, Group By | 2.3 |
| ConcreteRoleDescriptorService (181) | Project, Join, Group By | 2.1 |
| IterationService (103) | Project, Join, Group By | 2.0 |
| ParticipantService (266) | Project, Filter, Join, Group By | 1.6 |
| PhaseService (98) | Project, Join, Group By | 1.3 |
| RoleDao (15) | Project, Filter, Aggregation | 1.3 |

## 7 HAVING CLAUSE EXTRACTION

Thus far, we had deliberately set aside discussion of the HAVING clause. The reasons are that (a) $H_E$ predicates are difficult to extract due to close similarity to the filter predicates, $F_E$, and (b) Lemma 1, which guarantees a single-row input, is no longer valid. Nevertheless, we have been able to devise an efficient extraction technique modulo a few restrictions, the primary one being that the attribute sets in $F_E$ and $H_E$ are disjoint. However, incorporating this approach entails a significant reworking of the UNMASQUE pipeline, as well as modified algorithms for some of the extraction modules. Due to space limitations, we only summarize the $H_E$ extraction procedure here – the complete description is available in [19].

To accommodate HAVING, $G_E$ is identified immediately after $J_E$, and $F_E$ and $H_E$ are extracted subsequently. Leveraging the observation that $F_E$ constraints of the form $a \leq A \leq b$ can be equivalently re-written as $H_E$ predicates: $a \leq min(A)$ and $max(A) \leq b$, we initially extract both of them in a unified manner, followed by separation into the respective categories, as explained below.

**Predicate Extraction.** Each $H_E$ predicate is generically represented as $a \leq agg(A) \leq b$, which is separable into two components – $agg(A) \geq a$ and $agg(A) \leq b$ – that are individually identifiable. For the lower bound, we begin by sorting the table $t$ in ascending order on attribute $A$. Then, we progressively decrease the values in the column towards $i_{min}$, using a common logarithmic reduction function. Let $r$ be the first row in $t.A$ in which this reduction process results in an empty output before or upon reaching $i_{min}$. Now, based on the *location* of $r$, the possible aggregation functions can be refined. For instance, if $r$ is located in the interior of the table – i.e. not the first or last row – the aggregation is either $sum() \geq s_A$ or $avg() \geq a_A$, where $s_A$ (resp. $a_A$) is the current *sum* (resp. *avg*) of the values in column $A$. We differentiate these two cases by inserting a row with $t.A = 0$ since this keeps $sum()$ the same, but changes the $avg()$, and then recompute the output.

An analogous approach to the above can be applied to extract the upper bound (i.e. $agg(A) \leq b$). Finally, all $H_E$ predicates of the form $min(A) \geq a$ and $max(A) \leq a$ are converted to the corresponding filter predicates. This is to aid efficient query execution, since it is usually recommended to apply filters as early as possible.

## 8 COMPARISON WITH QRE

Although HQE is a conceptual variant of the long-standing QRE problem, UNMASQUE has been constructed from first principles. The reasons for doing so are explained here, using REGAL [25] as an exemplar for comparison.

The extraction pipeline sequence in REGAL identifies candidates for projections and joins *prior* to the other clauses. Whereas, UNMASQUE has to first identify the joins and filters so that the mutated databases in PROJECTION extraction, and the synthetic databases in the generation pipeline, are populated with s-values– that is, values compatible with the where clause predicates. Moreover, the generated databases have to be carefully constructed to produce *known*, albeit invisible, intermediate behavior.

Substantive differences also arise when we drill down to the individual operators. For instance, to construct the PROJECTION-JOIN component of the query, REGAL first generates candidate tables and projections using *value-based* comparisons between database columns and result columns. Next, it enumerates all feasible join graphs on the tables associated with a candidate. It then runs the set of queries corresponding to these derived candidates on $\mathcal{D}_I$, pruning those whose result does not match $\mathcal{R}_I$. This speculative process may lead to missing tables or spurious tables, depending on the $\mathcal{D}_I$ contents, as highlighted in [19]. In contrast, UNMASQUE uses a precise *error-based* identification of the query tables (Section 4.1) that has no dependence on $\mathcal{D}_I$ beyond requiring a populated result.

After identifying PROJECTION-JOIN candidates, REGAL creates a materialized view corresponding to each surviving candidate. On this view, a lattice of all possible grouping candidates is constructed, and a second round of candidate generation is done by incorporating aggregations. However, presence of unique values in any of the grouping columns may lead to incorrect selection of GROUP BY-AGGREGATION candidates, as seen in the example of Figure 2.

Turning to the FILTER clause, REGAL takes a *backward data-based* approach to its identification. Specifically, each aggregate value in the result is traced to its relevant view partition by constructing the contained tuples into a matrix whose dimensions are view attributes. An iterative process is used to select the matrix with the lowest dimensionality, and the minimum range limits on these dimensions, such that it is sufficient to produce the result. However, this could lead to missing dimensions and imprecise ranges, as highlighted in Figure 2(b). In contrast, UNMASQUE takes a *forward domain-based* constructive approach of using result cardinalities on extreme domain values to determine the presence of filters, followed by a binary search to obtain the precise constants.

Finally, since UNMASQUE can create any desired suite of input instances, it acquires the power to extract even complex clauses.

## 9 CONCLUSIONS AND FUTURE WORK

We introduced and investigated the problem of Hidden Query Extraction as a novel variant of QRE. Diverse HQE use-cases were outlined, ranging from database security to query rewriting, covering both explicit and implicit application opacity. Addressing the HQE problem proved to be challenging due to the inherent complexities of the query clauses, and the acute dependencies between them. As a first-cut solution, we presented UNMASQUE, which non-invasively and efficiently extracts a basal class of hidden SPJGHAOL queries through a pipeline that leverages database mutation and generation techniques to extract the clauses in a systematic manner.

A natural followup question is whether the extraction scope could be extended to a broader range of common SQL constructs. Based on our preliminary investigations, it appears that outer-joins, disjunctions and set operators could eventually be extracted under some restrictions. Nested queries, however, pose a formidable challenge that may require novel technology. More fundamentally, formally characterizing the extractive power of active learning techniques in the HQE context is an open theoretical problem.

## ACKNOWLEDGMENTS

We thank Ananya Appan and Manish Kesarwani for their technical contributions to our research, and S. Sudarshan for constructive feedback on the draft paper. This work was supported in part by a J C Bose Fellowship from SERB, Govt. of India, GCP grants from Google India, and a CSR grant from Huawei India.

# REFERENCES


[1] [n.d.]. ENKI Blogging App. https://github.com/xaviershay/enki
[2] [n.d.]. JOB Benchmark. https://github.com/gregrahn/join-order-benchmark
[3] [n.d.]. RUBiS Auction Site. https://projects.ow2.org/view/rubis
[4] [n.d.]. Software is Fragile. www.softwareheritage.org/mission/software-is-fragile
[5] [n.d.]. SQL Shield. www.sql-shield.com
[6] [n.d.]. Strings. https://docs.microsoft.com/en-us/sysinternals/downloads/strings
[7] [n.d.]. TPC-DS. www.tpc.org/tpcds/
[8] [n.d.]. TPC-H. www.tpc.org/tpch/
[9] [n.d.]. UCI Machine Learning Repository. https://archive.ics.uci.edu/
[10] [n.d.]. UNMASQUE Video. https://dsl.cds.iisc.ac.in/projects/HIDDEN/unmasque.mp4
[11] [n.d.]. Wilos Process Orchestration Software. www.openhub.net/p/6390
[12] A. Bonifati, R. Ciucanu, and S. Staworko. 2016. Learning Join Queries from User Examples. *ACM TODS* 40, 4 (2016).
[13] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. 2011. DBridge: A Program Rewrite Tool for Set-Oriented Query Execution. In *Proc. of ICDE Conf.*
[14] A. Cheung, A. Solar-Lezama, and S. Madden. 2013. Optimizing Database-Backed Applications with Query Synthesis. In *Proc. of PLDI Conf.*
[15] P. da Silva. 2019. *SQUARES:A SQL Synthesizer Using Query Reverse Engineering.* Master's thesis. https://web.ist.utl.pt/ist181151/81151-pedro-silva_dissertacao.pdf
[16] D. Kalashnikov, L. Lakshmanan, and D. Srivastava. 2018. FastQRE: Fast Query Reverse Engineering. In *Proc. of SIGMOD Conf.*
[17] G. Karvounarakis, Z. Ives, and V. Tannen. 2010. Querying Data Provenance. In *Proc. of SIGMOD Conf.*
[18] K. Khurana and J. Haritsa. 2020. UNMASQUE: A Hidden SQL Query Extractor. *PVLDB* 13, 12 (2020).
[19] K. Khurana and J. Haritsa. 2021. *Opaque Query Extraction.* Technical Report. Indian Institute of Science. https://dsl.cds.iisc.ac.in/publications/report/TR/TR-2021-02.pdf
[20] L. Lazar and E. Erez. 2018. A Deep Dive into Database Attacks [Part I]: SQL Obfuscation. www.imperva.com/blog/database-attacks-sql-obfuscation
[21] K. Panev and S. Michel. 2016. Reverse Engineering Top-k Database Queries with PALEO. In *Proc. of EDBT Conf.*
[22] K. Ramachandra, K. Park, K. Emani, A. Halverson, C. Galindo-Legaria, and C. Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *PVLDB* 11, 4 (2017).
[23] S. Shah, S. Sudarshan, S. Kajbaje, S. Patidar, B. Gupta, and D. Vira. 2011. Generating test data for killing SQL mutants: A constraint-based approach. In *Proc. of ICDE Conf.*
[24] W. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava. 2017. Reverse Engineering Aggregation Queries. *PVLDB* 10, 11 (2017).
[25] W. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava. 2018. REGAL+: Reverse Engineering SPJA Queries. *PVLDB* 11, 12 (2018).
[26] Q. Tran, C. Chan, and S. Parthasarathy. 2014. Query Reverse Engineering. *The VLDB Journal* 23, 5 (2014).
[27] J. Tuya, M. Cabal, and C. Riva. 2010. Full predicate coverage for testing SQL database queries. *STVR* 20, 3 (2010).
[28] C. Wang, A.Cheung, and R. Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proc. of PLDI Conf.*
[29] M. Watson. 2019. View SQL Queries From Your Code With Prefix. www.stackify.com/view-sql-with-prefix/
[30] M. Zhang, H. Elmeleegy, C. Procopiuc, and D. Srivastava. 2013. Reverse Engineering Complex Join Queries. In *Proc. of SIGMOD Conf.*