

UNMASQUE: A Hidden SQL Query Extractor

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

ABSTRACT

Given a database instance and a populated result, query reverse-engineering attempts to identify candidate SQL queries that produce this result on the instance. A variant of this problem arises when a ground-truth is additionally available, but hidden within an opaque database application. In this demo, we present UNMASQUE, an extraction algorithm that is capable of precisely identifying a substantive class of such hidden queries. A hallmark of its design is that the extraction is completely non-invasive to the application. Specifically, it only examines the results obtained from application executions on databases derived with a combination of data mutation and data generation techniques, thereby achieving platform-independence. Further, potent optimizations, such as database size reduction to a few rows, are incorporated to minimize the extraction overheads. The demo showcases these features on both declarative and imperative applications.

PVLDB Reference Format:

Kapil Khurana and Jayant R. Haritsa. UNMASQUE: A Hidden SQL Query Extractor. *PVLDB*, 13(12): 2809-2812, 2020.
DOI: <https://doi.org/10.14778/3415478.3415481>

1. INTRODUCTION

Over the past decade, query reverse-engineering (QRE) has attracted considerable research attention. The generic problem tackled here is the following: Given a database instance \mathcal{D} and a populated result \mathcal{R} , identify a candidate SQL query Q_c such that $Q_c(\mathcal{D}) = \mathcal{R}$. Impressive progress has been made on addressing the QRE problem, with potent tools such as Talos[3] and Regal[2] having come to the fore.

A variant of the QRE problem, recently introduced in [1], arises when a *ground-truth* query is additionally available, but in a hidden form that is not easily accessible. In this variant, termed *hidden-query extraction* (HQE), the objective is defined as: *Given a black-box application \mathcal{A} containing a hidden SQL query Q_H , and a database instance \mathcal{D} on which Q_H produces a populated result \mathcal{R} , unmask Q_H to reveal the original query.* Such “hidden-executable” situations can arise in a variety of real-world contexts, including: (i) encrypted or obfuscated database applications; (ii) legacy code

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 13, No. 12

ISSN 2150-8097.

DOI: <https://doi.org/10.14778/3415478.3415481>

whose source has been lost over time; (iii) presence of third-party proprietary tools in the workflow; and (iv) poorly documented software inherited from external developers.

The presence of the hidden ground-truth can be leveraged to provide a variety of advantages relative to the earlier QRE work, including: (i) Independence from the $(\mathcal{D}, \mathcal{R})$ instance; (ii) Precise identification of query constants (e.g. via binary search on attribute domains); (iii) Extraction of advanced constructs (e.g. LIKE, LIMIT), and (iv) Extraction efficiency on large databases. Moreover, using QRE techniques to provide a “seed query” for HQE is not a viable approach since (a) their candidate queries may differ from the original query in virtually all the constructs, and (b) they do not support a modular extraction of component clauses. Therefore, the HQE procedures have to be designed from scratch.

We took a first step towards addressing the HQE problem in [1]. Specifically, we presented UNMASQUE¹, an algorithm that uses a judicious combination of *database mutation* and *synthetic database generation* to precisely identify the hidden query. Currently, UNMASQUE is capable of extracting a restricted but substantive class of SPJGAOL (SELECT, PROJECT, JOIN, GROUPBY, AGGREGATE, ORDERBY, LIMIT) queries. As an exemplar, consider Q_H in Figure 1, which encrypts a TPC-H Q3-based query in a stored procedure, and features all these clauses. UNMASQUE was able to successfully extract all semantic aspects of this query, as shown in the UNMASQUE output screenshot of Figure 6.

```
Create Procedure tpch_HQ with Encryption BEGIN
Select      l_orderkey, sum(l_extendedprice) as revenue,
            o_orderdate, o_shippriority
            customer, orders, lineitem
From
Where      c_custkey = o_custkey and l_orderkey = o_orderkey
            and c_mktsegment = 'BUILDING'
            and o_orderdate < date '1995-03-15'
            and l_shipdate > date '1995-03-15'
group by   l_orderkey, o_orderdate, o_shippriority
order by   revenue desc, o_orderdate
limit      10;
END
```

Figure 1: Hidden Query Example (Q_H)

A hallmark of UNMASQUE’s design is that the extraction is completely non-invasive with respect to the application code, examining only the results obtained from its executions on a variety of carefully constructed databases. This makes the tool immediately usable, either in lieu of, or prior to, invoking stronger forensic tools. Moreover, it makes the tool essentially platform-independent with regard to the underlying database engine.

¹Unified Non-invasive MACHine for Sql QUery Extraction

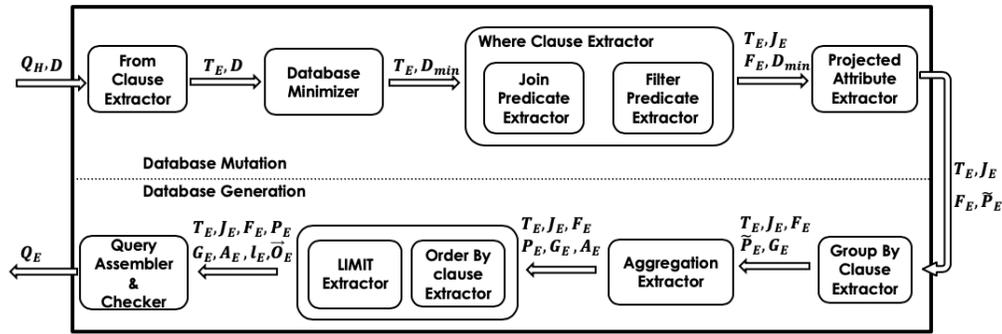


Figure 2: UNMASQUE Architecture

Finally, to ensure extraction efficiency, UNMASQUE incorporates size-reduction optimizations which ensure that all extraction procedures operate on minuscule databases containing only a handful of rows. For instance, the initial database \mathcal{D} is whittled down to the smallest subset D_{min} on which Q_H continues to provide a populated result before applying our mutation techniques. In fact, for UNMASQUE’s current extraction query scope, D_{min} provably contains only a *single row* in each of the relevant database tables.

In analogous fashion, the synthetically created databases are also carefully designed to be very thinly populated. For instance, the GROUP BY columns are identified using a database with a maximum of just three rows in each query-related table.

Thanks to these optimizations, the extraction of the example query in Figure 1 was completed within *ten minutes* on a vanilla computing platform hosting a 100 GB version of the TPCB database. Moreover, even when the initial database was scaled by an order of magnitude to 1 TB, the extraction was completed in less than twenty minutes. A detailed performance evaluation over representative TPCB-based queries is described in [1].

Extraction Workflow. UNMASQUE operates according to the pipeline shown in Figure 2, where it unmaskes the hidden query elements in a structured manner, starting with the FROM clause, continuing on to the JOIN and FILTER predicates, following up with the PROJECTION, GROUP BY, AGGREGATION columns, and concluding with the ORDER BY and LIMIT functions. The initial SPJ elements are extracted using database mutation strategies, whereas the subsequent GAOL elements are extracted leveraging database generation techniques. The final component is the QUERY ASSEMBLER which puts together the different elements of Q_E and performs canonification to ensure a standard output format.

Demo Highlights. Our objective in the demo is to visually and interactively showcase the query extraction features of UNMASQUE, which is implemented in about 10K lines of Python code. The user interfaces, detailed in Section 3, include: (a) **Database Instance Selection**, where the user can choose the database instance on which to evaluate query extraction; (b) **Hidden Query Input**, through which the user can submit an opaque database logic, either via direct SQL input that is subsequently encrypted, or via an imperative logic executable; (c) **Extraction Pipeline**, where users can follow the step-by-step extraction process, including viewing the tiny databases used in each of these modules; and, finally (d) **Extraction Output**, which presents the extracted query along with statistical profiles of the extraction time, number of mutations and executable invocations, etc., as well as a textual comparison of the extracted and hidden queries.

2. DESIGN OF UNMASQUE

Currently, UNMASQUE assumes that the application contains either a single SQL query, or imperative logic that is expressible in a single query. Further, the coverage is limited to a restricted but substantive class of SPJGAOL queries. The primary assumptions are that the query is nesting-free, disjunction-free and function-free; further, all join predicates are inner equi-joins between key columns, and all filter predicates are on non-key columns. Interestingly, we support the LIKE operator, including wildcards, for filtering textual columns. The complete details of UNMASQUE, including formal proofs of its ability to extract complex queries, are available in [1].

To set up UNMASQUE’s extraction process, we create a silo in the database that has the same table schema as the original user database. Subsequently, all referential integrity constraints are dropped from the silo tables, since the extraction process requires the ability to construct alternative database scenarios that may not be compatible with the existing schema. We then create the following template representation for the to-be-extracted query Q_E :

```

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

```

and sequentially identify each of the constituent elements, as per the pipeline of Figure 2, described below.

2.1 Mutation Pipeline

The first half of the pipeline, referred to as Mutation Pipeline (MP), is based on mutations of the original database \mathcal{D} to obtain T_E (tables in FROM clause), followed by reduction to D_{min} , and then mutations of this reduced database to obtain the SPJ elements that deliver the raw query results. MP implements the mutations by making targeted changes to a specific table or column while keeping the rest of D_{min} intact.

We illustrate this process through the identification of join predicates, J_E , between primary-keys (pk) and foreign-keys (fk) for acyclic database schema graphs. First, we create a *candidate join-graph* (CJG), whose vertices are the key columns in T_E , and whose edges are all *schematically* permissible pk-fk joins between these columns. Then, we iteratively check each edge e in the CJG for its presence in the hidden query. For this evaluation, D_{min} is mutated to D_{mut} such that only the join predicate corresponding to e is *not* satisfied. This is achieved by identifying the two connected components obtained after removing e from CJG, and mutating the key column values in one of the components such that the new values are different to all the key values in the other component. Finally, we observe the result of Q_H on D_{mut} – an empty (resp. populated) result implies the presence (resp. absence) of e in the hidden query’s join graph.

For instance, to check the join edge $\langle c_custkey, o_custkey \rangle$ in Figure 1, we construct D_{mut} by *negating* the $c_custkey$ values in D_{min} , thereby ensuring the join predicate is not satisfied. After that, we run Q_H on this D_{mut} instance, and the empty output implies the edge is present in Q_H 's join graph.

2.2 Generation Pipeline

The second half of the pipeline, referred to as Generation Pipeline (GP), is based on the generation of carefully-crafted synthetic databases. It caters to the GAOL query clauses, which manipulate the raw SPJ results. The modules in this segment require generation of new data for all the query-related tables under various row-cardinality and column-value constraints. We deliberately depart from the mutation approach here since these constraints may not all be satisfied by the original database instance.

We illustrate this process through the identification of the group-by columns, G_E . For instance, to verify grouping on a generic table column $t.A$, we create a D_{gen} such that the *intermediate* result produced by the SPJ part of Q_H has exactly **3 rows**, which satisfy the following condition: $t.A$ has a common value in only two rows, while all other columns have the same value in all three rows. Now, if the *final* result of applying Q_H on D_{gen} contains **2 rows**, it implies that the grouping is only due to the two different values in $t.A$, making it part of G_E .

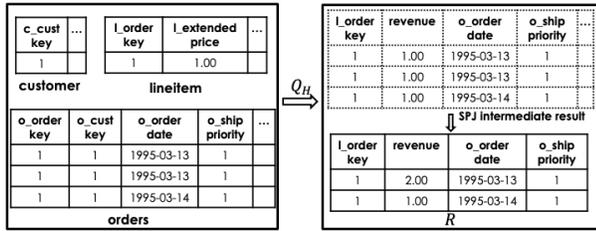


Figure 3: D_{gen} for Checking Grouping on $o_orderdate$ (Q_H)

The key to the above test is the creation of a suitable D_{gen} such that: (i) all the filter and join predicates are satisfied; (ii) all remaining attributes in T_E other than $t.A$ are assigned a single value in all the rows; and (iii) $t.A$ is assigned the same value in two rows and a different value in the third row. A constructive procedure for generating such D_{gen} is explained in [1] – as a concrete example, the D_{gen} for checking the presence of $o_orderdate$ in G_E of the example query in Figure 1 is shown in Figure 3. Here, the **ORDERS** table features 3 rows while the remaining tables, **LINEITEM** and **CUSTOMER**, have one row apiece. The output, **R**, having two rows indicates that $o_orderdate$ is part of G_E .

3. UNMASQUE DEMONSTRATION

In the demo, the audience will actively engage with a variety of visual scenarios that showcase the functionality and utility of UNMASQUE. As samples, the tool's interfaces at various stages of extraction are shown in Figures 4 through 6, and their contents are explained below. An illustrative video of UNMASQUE in operation is available at the project website [4].

Database Instance Selection. The top pane in Figure 4 profiles the database connectivity inputs for selecting specific relational engines and database instances. In the demo, we will showcase extraction on the TPC-H and TPC-DS benchmark databases at various sizes, ranging from 1 GB to 1 TB, hosted on the PostgreSQL and Microsoft SQL Server platforms.

Hidden Query Input. The bottom left and right panes in Figure 4 provide the alternative input methods to UNMASQUE. In the first method, the user directly types an SQL query in the template form, and then clicks the **Hide Query** button, resulting in an encrypted or obfuscated stored procedure. A sample encryption on the SQL Server platform was shown in Figure 1.

The second user option is to invoke a (predefined) opaque stored procedure, or an opaque executable with embedded imperative database logic. The query extraction here will showcase how UNMASQUE offers a light-weight and platform-independent approach to imperative-to-declarative translation. Further, it is usable even when only the executable, and not the source, is available.

Extraction Pipeline Progress. Once the extraction process is started, UNMASQUE displays the screen of Figure 5. The upper pane presents the mutation and generation pipelines, and the lower shows the extracted query template, which is step-by-step fleshed out as the extraction process proceeds through these pipelines.

Initially, all modules in the pipeline are colored *Orange*. Once the execution phase reaches a module, it is coloured *Yellow*, and subsequently to *Green* after completion; further, its output is populated in the corresponding clause of the query template. The snapshot shown in Figure 5 marks the transition point where all the modules in Mutation Pipeline have completed (resulting in identification of the SPJ elements), and the Generation Pipeline execution has just begun.

Internals of Extraction Modules. Post query extraction, users can drill-down into each module and view the specific databases used for extraction (similar to Figure 3). Further, the data structures used in the module are also shown, with some having visual interfaces for interactively testing the constructs (e.g. candidate join graphs). These features provide direct insights into the mutation and generation algorithms used for the extraction.

Extraction Output Analysis. After completing all the pipeline modules, UNMASQUE displays the final extracted query, as shown in Figure 6. This is accompanied with a statistical performance profile that includes the total and component-wise *extraction times*, number of *executable invocations*, and related information. For instance, the total time for extracting the example query was 44 seconds, and required 202 invocations of the application executable. Users can also verify the correctness of the extraction by running the **CHECKER** module via the **Run Checker** button which supports the regression test suite of the original database.

Acknowledgements

This work was supported in part by a J. C. Bose Fellowship from Dept. of Science and Technology, Govt. of India, and a GCP grant from Google India.

4. REFERENCES

- [1] K. Khurana and J. Haritsa. Hidden Query Extraction. *Tech. Rep. 2020-01, DSL, IISc*. dsl.cds.iisc.ac.in/publications/report/TR/TR-2020-01.pdf
- [2] W. Tan, M. Zhang, H. Elmeleegy and D. Srivastava. REGAL⁺: Reverse Engineering SPJA Queries. *PVLDB*, 11(12):1982–1985, Aug. 2018.
- [3] Q. Tran, C. Chan and S. Parthasarathy. Query Reverse Engineering. *The VLDB Journal*, 23(5):721-746, Oct. 2014.
- [4] dsl.cds.iisc.ac.in/projects/HIDDEN

UNMASQUE

DATABASE CONNECTIVITY

DB engine: DB Instance: Connection Parameters:

DIRECT HIDDEN QUERY INPUT		EXTERNAL EXECUTABLE INPUT	
SELECT	<input type="text" value="l_orderkey, sum(l_extendedprice) as revenue, o_orderdate, o_shippriority"/>	* Stored Procedure	
FROM	<input type="text" value="customer, orders, lineitem"/>	Stored Procedure: <input type="text" value="tpch_HQ"/>	
WHERE	<input type="text" value="c_custkey = o_custkey and l_orderkey = o_orderkey and c_mktsegment = 'BUILDING' and o_orderdate < date '1995-03-15' and l_shipdate > date '1995-03-15'"/>	<input type="checkbox"/> Imperative Executable	
GROUP BY	<input type="text" value="l_orderkey, o_orderdate, o_shippriority"/>	File Path: <input type="text"/>	
ORDER BY	<input type="text" value="revenue desc, o_orderdate"/>	Execution Command: <input type="text"/>	
LIMIT	<input type="text" value="10"/>	<input type="button" value="Hide Query"/>	

Save Extracted Query

Figure 4: UNMASQUE Input Screen

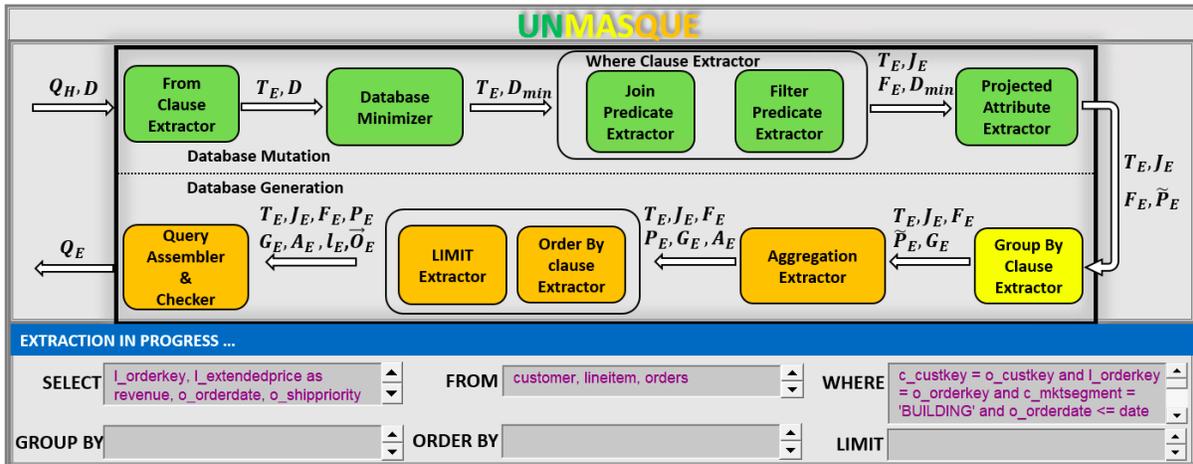


Figure 5: UNMASQUE Extraction Visualization (Green: executed; Yellow: executing; Orange: awaiting execution)

UNMASQUE

HIDDEN QUERY	EXTRACTED QUERY
SELECT: <input type="text" value="l_orderkey, sum(l_extendedprice) as revenue, o_orderdate, o_shippriority"/>	SELECT: <input type="text" value="l_orderkey, sum(l_extendedprice) as revenue, o_orderdate, o_shippriority"/>
FROM: <input type="text" value="customer, orders, lineitem"/>	FROM: <input type="text" value="customer, lineitem, orders"/>
WHERE: <input type="text" value="c_custkey = o_custkey and l_orderkey = o_orderkey and c_mktsegment = 'BUILDING' and o_orderdate < date '1995-03-15' and l_shipdate > date '1995-03-15'"/>	WHERE: <input type="text" value="c_custkey = o_custkey and l_orderkey = o_orderkey and c_mktsegment = 'BUILDING' and o_orderdate <= date '1995-03-14' and l_shipdate >= date '1995-03-14'"/>
GROUP BY: <input type="text" value="l_orderkey, o_orderdate, o_shippriority"/>	GROUP BY: <input type="text" value="l_orderkey, o_shippriority, o_orderdate"/>
ORDER BY: <input type="text" value="revenue desc, o_orderdate"/>	ORDER BY: <input type="text" value="revenue desc, o_orderdate asc"/>
LIMIT: <input type="text" value="10"/>	LIMIT: <input type="text" value="10"/>

DB Instance: TPC-H 10 GB Total Extraction Time: 44 sec
 Clauses with Syntactic Differences: From, Where, Group By Number Of Executable Invocations: 202

Figure 6: UNMASQUE Final Output Screen