Opaque Query Extraction*

Kapil Khurana Jayant Haritsa

Technical Report
TR-2021-02
(March 2021)

Database Systems Lab
Indian Institute of Science
Bangalore 560012, India

`http://dsl.cds.iisc.ac.in`

This report is a significantly extended and expanded version of **TR-2020-01**.

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, such as table minimization and sampling, are incorporated to reduce extraction overheads. A detailed evaluation over applications hosting hidden SQL queries, or their imperative versions, demonstrates that UNMASQUE correctly and efficiently extracts these queries.

1 Introduction

Over the past decade, *query reverse-engineering* (QRE) has evinced considerable interest from both the database and programming language communities (e.g. [29, 25, 23, 22, 11, 4, 1, 16, 5, 27]). 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 $\mathcal{Q}_c$ such that $\mathcal{Q}_c(\mathcal{D}_I)= \mathcal{R}_I$.* 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) explanation for unexpectedly missing tuples in the result.

```
Create Procedure tpch_HQ with Encryption  BEGIN
Select  l_orderkey, sum(l_extendedprice * (1 - l_discount))
          as  revenue, o_orderdate, o_shippriority
From      customer, orders, lineitem
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
```

(a) **Hidden Query ($\mathcal{Q}_H$)**

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

(b) **UNMASQUE Output Query ($\mathcal{Q}_E$)**

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

```
select  l_orderkey, o_orderdate,
           o_shippriority,
           sum(l_extendedprice)
           as  revenue
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) **Sample REGAL Input Query**

```
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) **Sample REGAL Output Query**

Figure 2: REGAL QRE Example

Impressive progress has been made on addressing the QRE problem, with the development of elegant tools such as TALOS [25], REGAL [23] and SCYTHE [27]. Notwithstanding, there are intrinsic challenges underlying the problem framework: First, the output query $\mathcal{Q}_c$ 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 additionally 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 effectively 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 $\mathcal{Q}_H$ (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 $\mathcal{Q}_H$ 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 $\mathcal{Q}_H$ such that $\mathcal{Q}_H(\mathcal{D}_i) = \mathcal{R}_i \; \forall i$.

The presence of the hidden ground-truth can be leveraged to deliver 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 $\mathcal{Q}_H$; (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 [25]); (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 be used to provide a *seed query* for HQE, followed by refinements to precisely identify the hidden query. However, as explained in Section 9, this is not practical for both conceptual and performance reasons.

Therefore, we have approached the HQE problem from *first principles*. Our experience is that it proves to be 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 aggregration functions, and (d) presence of computed column functions.

UNMASQUE Extractor

We take a first step towards addressing the HQE problem here by presenting UNMASQUE[1], an algorithm that exposes the hidden query $\mathcal{Q}_H$ 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

[1]Unified Non-invasive MAchine for Sql QUery Extraction

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 – specifically, single-block equi-join conjunctive queries expressible in the form (as per the notation in Table 1):

Select (P_E, A_E) **From** T_E **Where** $J_E \wedge F_E$

Group By G_E **Having** H_E **Order By** $\overrightarrow{\mathcal{O}_E}$ **Limit** l_E

The specific query coverage and underlying assumptions are enumerated in Section 3.

As an exemplar of a non-trivial query that falls in the ambit of its extraction scope, consider $\mathcal{Q}_H$ in Figure 1a – 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, $\mathcal{Q}_E$, is shown in Figure 1b – 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, it is unable to handle the revenue function, ORDER BY and LIMIT constructs. Secondly, consider the much simpler query version shown in Figure 2a. Even in this case, REGAL produces the output shown in Figure 2b – while the tables and joins are detected correctly, significant discrepancies exist in the filters, grouping columns and aggregation functions. Moreover, as explained in Section 9, errors could arise in the tables and the joins as well, depending on the specific ($\mathcal{D}_I$,$\mathcal{R}_I$) instance supplied to the tool. Finally, producing this limited outcome itself took considerable time and resources on a well-provisioned platform.

Extraction Workflow

UNMASQUE operates according to the pipeline shown in Figure 3, where it gradually extracts the hidden query elements in a structured and 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). [3]

The initial elements (SPJ) are extracted using *database mutation* strategies, whereas the subsequent ones (GAOL) leverage *database generation* techniques. Most of the modules feature carefully crafted methods for unambiguous identification. The final pipeline component is the ASSEMBLER which combines the various elements of $\mathcal{Q}_E$ and performs canonification to obtain a standard output format. Further, it implements various automated checks to verify correctness of extraction. A demo version of UNMASQUE was presented in [13], and a video of UNMASQUE in operation is available at [54].

Extraction Efficiency

To cater to extraction efficiency concerns, 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 $\mathcal{Q}_H$ 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.*

At first glance, it may appear that $\mathcal{D}_{min}$ can be easily obtained using well-established provenance techniques (e.g. [12]). However, due to the hidden nature of $\mathcal{Q}_H$, these approaches are no longer

[2]SELECT, PROJECT, JOIN, GROUPBY, HAVING, AGG, ORDER, LIMIT

[3]In Figure 3, the module pairs shown in common boxes can be processed independently.

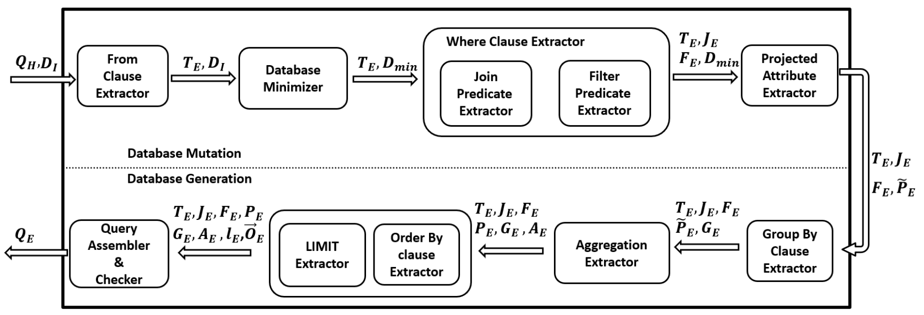


Figure 3: UNMASQUE Architecture

viable. 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 queries arising in synthetic (TPC-H, TPC-DS) and real (JOB on IMDB dataset, UCI) environments, and (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*, which is reasonable given that its native execution takes around 5 minutes on the same platform.

Organization

The rest of the report is organized as follows: In Section 2, a variety of extraction use-cases are outlined. Then, in Section 3, a precise description of the HQE problem framework is provided, along with the associated notations. Sections 4 and 5 present the components of the UNMASQUE pipeline, which progressively reveal different facets of the hidden query. The experimental framework and performance results are reported in Section 6. Extraction of the HAVING clause is discussed in Section 7, while a few other extensions are summarized in Section 8. Related work is reviewed in Section 9. Finally, our conclusions and future research avenues are summarized in Section 10.

2 HQE Deployment Scenarios

We now discuss deployment scenarios where HQE techniques could be of utility. Our assumption is that the extraction process is invoked by the owner, or a privileged user, of the underlying database, possessing both read and write rights on the contents. The opaque application may also be owned by the same person, or submitted by an external source.

As mentioned in the Introduction, there are two opacity scenarios – the first, where the application is explicitly opaque due to encoding, and the other, where it is implicitly opaque due to representational complexity. We present below sample use-cases for both scenarios.

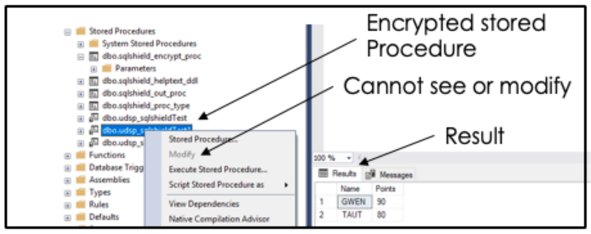
2.1 Explicit Opacity

Only the executable object code of the application is available here, and therefore, identifying the underlying interactions with the database system can help to determine the data processing logic embedded in the application.

Recovering Lost SQL It often happens, especially with legacy industrial applications, that with the passage of time, the original source code becomes lost [47]. However, to understand the output, we may need to establish the logic connecting the database input to the observed result. Second, 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 [40]). 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 [48] offers encryption of stored SQL procedures on Microsoft SQL Server platforms.

An annotated screenshot of this tool's interface is shown in Figure 4, highlighting the opacity of the SQL procedure post-encryption.



	Name	Points
1	GWEN	90
2	TAUT	80

Figure 4: Encrypted Stored Procedure

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, the SQL Shield tool 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. Cloud), the hidden query may not be easily extractable – it is precisely such impenetrable scenarios that motivate our study.

Enhancing Database Security. Database servers are often preferred targets of sophisticated hacking attacks (e.g. [32, 31]), and therefore security checks for suspicious SQL statements are omnipresent in database firewalls and activity monitoring tools. However, these checks are easily defeated through obfuscation techniques. For instance, even a simple HEX encoding is often sufficient to get through the defenses – as a case in point, *"select * from passwords"* would appear as an incomprehensible *"73656C656374202A2066726F6D2070617373776F726473"* to the policing tools after this encoding [34].

In this context, given a hidden query that has been prematurely stopped, say due to concerns about its origins, the database owner can offline use a HQE tool to ascertain the real intent of the application so that further steps can be taken, if required, to secure the system. UNMASQUE takes the first step towards establishing a new approach to determining application objectives through query extraction, rather than directly resorting to heavy-weight techniques based on system audits and code forensics.

2.2 Implicit Opacity

We now move on to situations where the application source is available, but the complex internal representation makes it hard to comprehend for the new developer or the database engine. This is especially true for machine-generated mappings, as well as hand-me-down programs sourced from third parties.

Learning-based Query Rewriting The use of automated ORM tools is common today, often resulting in unintelligible SQL formulations [53]. There are two problems that 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, in practice, the plans generated are usually significantly slower. However, it may be hard to apply these rules on convoluted original representations. Modeling the rewriting as a HQE problem, however, can help to achieve the objective since only databases and results are involved, and not the query itself, eventually leading to a "lean" equivalent. In this sense, HQE can also be viewed as a *learning-based* rewriting technique. Of course, the approach is viable only for "canned queries" where an initial compile-time investment on extraction can be amortized over subsequent executions.

As a concrete example of this use-case, consider the ORM-generated query shown in Figure 5a. The Postgres optimizer finds it hard to parse this complex representation and throws up a plan that is unable to effectively use the filters on ORDERS and LINEITEM. In contrast, UNMASQUE extracts the "textbook query" shown in Figure 5b, and the Postgres plan for this clean formulation runs an *order-of-magnitude faster* than the original plan.

```
Select (Select Sum (Case When (Exists ( Select 1 as [C1]
Where ([E1].[o_orderstatus] = 'O' and [E2].[l_shipdate] > '1998-04-03')))
Then 1 Else 0 End) as [C1]
From [dbo].[Orders] as [E1] Inner Join [dbo].[Lineitem] as [E2]
On [E1].[o_orderkey] = [E2].[l_orderkey]) as SingleValueAnswer
From (Select 1 as X) as [Y]
```

(a) **ORM generated query**

```
Select count(*)
From Orders, Lineitem
Where Orders.o_orderkey = Lineitem.l_orderkey and
Orders.o_orderstatus = 'O' and Lineitem.l_shipdate > '1998-04-03';
```

(b) **Equivalent SQL extracted by UNMASQUE**

Figure 5: Learning-based Query Rewriting

Lightweight Code Conversion Often, for programming convenience or due to lack of SQL expertise, software developers may choose to write *imperative code*. This may lead to serious execution inefficiencies due to not leveraging the database system's potent optimization abilities – for instance, the use of indexes. 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 [18] in SQL Server). However, these tools are host-language-specific (e.g. TSQL in the case of Froid), and require support for special operators (e.g. APPLY or LATERAL) which are not 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 application and platform-independent.

3 Problem Framework

We assume that an application executable object file is provided, which contains either a single SQL query or imperative logic that can be expressed in a single query. If there are multiple queries in the application, we assume that each of them is invoked with a separate function call, and not batched together, reducing to the single query scenario. This assumption is consistent with open source projects such as *Wilos* [52], which contain code segments wherein each function implements the logic of a single relational query.

If the hidden SQL query is present as-is in the executable, it can be trivially extracted using standard string extraction tools (e.g. *Strings* [40]). However, if there has been post-processing, such as *encryption* or *obfuscation*, for protecting the application logic, this option is not feasible. An alternative strategy is to re-engineer the query from the *execution plan* at the database engine. However, this knowledge is also often not accessible – for instance, the SQL Shield tool[48] blocks out plan visibility in addition to obfuscating the query. Finally, if the query has been expressed in imperative code, then neither approach is feasible for extraction.

Moving on to the database contents, there is no inherent restriction on column data types, but we assume for simplicity, the common *numeric* (int, bigint and float with fixed precision), *character* (char, varchar, text), *date* and *boolean* types. The database is freely accessible through its API, supporting all standard DML and DDL operations, including creation of a test silo for extraction purposes.

3.1 Extractable Query Class

The QRE literature has primarily focused on constructing generic SPJGA queries featuring only key-based equi-joins and not supporting nesting, disjunctions or set operators. We share these basic structural restrictions, but our extraction scope is significantly enlarged, including HOL constructs, LIKE comparators, and multi-linear scalar functions on the columns. To achieve this extended coverage, we require some additional mild assumptions. The specific restrictions on the supported queries are the following:

1. Each table in the database schema appears at most once in the FROM clause. Further, the join graph is a subgraph of the schema graph (comprised of all valid PK-FK and FK-FK edges). The join predicates are inner equi-joins between keys.

2. The query is a single monolithic block without any nesting. We support multi-linear scalar column functions, but arbitary computational expressions should not be present.

3. The WHERE clause is composed of a pure *conjunction* of filter and join predicates.

4. All filter predicates feature only non-key columns. For numeric columns, arithmetic selections of the type *column op value* where *op* is from $=, \leq, \geq, <, >, between$ are supported, whereas for textual columns, the *like* operator is extractable. In addition, boolean and nullity predicates are also managed.

5. All aggregate functions are from the standard SQL constructs: *min(), max(), count(), sum(), avg()*.

6. The columns in the ORDER BY clause are all from the set of projected columns, which is typically the case in practice.

7. The limit value is at least 3.

We hereafter refer to the above class of supported queries as *Extractable Query Class* (EQC).

Due to the difference in extraction frameworks, we initially present UNMASQUE only for SPJGAOL queries, deferring the HAVING clause to Section 7. Accordingly, we qualify EQC as EQC^{-H} (EQC

Symbol	Meaning	Symbol	Meaning (wrt query $\mathcal{Q}_E$)
$\mathcal{A}$	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
T	Set of all Tables in $\mathcal{D}_I$	F_E	Set of Filter predicates
SG	Schema Graph of $\mathcal{D}_I$	G_E	Set of Group By columns
$\mathcal{Q}_H$	Hidden Query	H_E	Set of Having predicates
$\mathcal{Q}_E$	Extracted Query	l_E	Limit value
D_{min}	Reduced Database	P_E	Set of native Projections with mapped result columns
D^t	Database with at most t rows in tables of $\mathcal{Q}_E$	A_E	Set of Aggregations with mapped result columns
D_{mut}	Mutated database	$\overrightarrow{\mathcal{O}_E}$	Sequence of Ordered result columns
D_{gen}	Generated database	t_D	Average running time of the query on database D
		S_E	Total number of attributes in the tables in T_E

Table 1: Notations

without HAVING) in Sections 4 through 6. Further, 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 at the end.

The notations used in our description of the extraction pipeline are summarized in Table 1. To highlight its black-box nature, the application executable is denoted by $\mathcal{E}$, while $\overrightarrow{\mathcal{O}_E}$ has a vector symbol to indicate that the ordering columns form a *sequence*.

3.2 Overview of the Extraction Approach

To set up the extraction process, we begin by creating 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 $\mathcal{Q}_E$:

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

and sequentially identify each of the constituent elements, as per the pipeline shown in Figure 3.

The initial segment of the pipeline is based on mutations of the original/reduced database and is responsible for handling the SPJ features of the query which deliver the raw query results. The modules in this segment require targeted changes to a specific table or column while keeping the rest of the database intact.

In contrast, the second pipeline segment is based on the generation of carefully-crafted synthetic databases. It caters to the GAOL query clauses, which are based on manipulation of the raw 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 be satisfied by the original database instance.

At an intuitive level, the common theme across the SPJ extraction algorithms is that *atomic* changes are made to individual database columns and the effect on the result is observed – with regard to change in cardinality change (Select, Join) or change in column value (Projection) – to establish presence in the respective clauses. 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. This calibrated construction process supports precise establishment of associations between the input database columns and the output result columns.

We hereafter refer to these two segments as the *Mutation Pipeline* and the *Generation Pipeline*, respectively, and present them in detail in the following sections.

Further, for the efficiency analysis, we assume a simple cost model, defined as follows: Let $|T|$ denote the size of table T, measured in terms of the row-cardinality. Then, the time to run a query which includes m tables (say $T_1, T_2, ..., T_m$) is directly proportional to the product of the table sizes, i.e. $|T_1| * |T_2| * ... * |T_m|$, or $\prod_{i=1}^{m} |T_i|$.

We will use τ to indicate the time taken by the application to operate on the original database $\mathcal{D}_I$ and produce the $\mathcal{R}_I$ result – with the above model, $\tau = c * \prod_{i=1}^{|T_E|} |T_i|$, where c is the proportionality constant.

4 Mutation Pipeline

The SPJ core of the query, corresponding to the FROM (T_E), WHERE (F_E, J_E) and SELECT (P_E) clauses, is extracted in the Mutation Pipeline segment of UNMASQUE. Aggregation columns in the SELECT clause are only identified as projections here, and subsequently refined to aggregations in Generation Pipeline.

4.1 From Clause

To identify whether a base table t is present in $\mathcal{Q}_H$, 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, we conclude that t is not in the query and the ongoing execution is terminated after a short timeout period. Finally, $temp$ is reverted to its original name t. By doing this check iteratively over all the tables in the schema, T_E is identified. For **Q3**, the procedure results in T_E ***= {customer, lineitem, orders}***.

The above "execution-with-error" approach is very quick. However, it may not always be feasible since either (i) the database engine may not be designed to issue such alerts, or (ii) the application may handle the error internally and not propagate the message to the user. We have therefore designed an alternative platform-agnostic "execution-with-zero-result" approach that is based on the following observation: Given our EQC, where only inner equi-joins are permitted, if any table in the FROM clause is empty, the query result will also be empty. So, we take each candidate table t in turn, rename it to $temp$, then create a new empty table t with the same schema as $temp$. Let this modified database be called D_{mut}. Subsequently, $\mathcal{E}$ is run on D_{mut} and the result is observed – if empty, t belongs to T_E. After that, table t is dropped and $temp$ is renamed to t – this transforms D_{mut} back to the initial database instance.

We hereafter use $\mathcal{D}_I$ to refer not to the entire original database, but specifically to the part relevant to $\mathcal{Q}_E$ – i.e. the contents of T_E.

4.1.1 Time Complexity

In the above "execution-with-error" approach, the executable invocation takes constant time as there is a predefined time limit on the execution time after which the query execution is interrupted. Hence, the time complexity of FROM clause extraction using "execution-with-error" approach is $O(|T|)$ since all the Ttables in the schema have to be iteratively checked for their presence in the query.

However, in the "execution-with-zero-result" approach, each execution invocation might take τ in the worst case. Hence, the time complexity of FROM clause extraction using "execution-with-zero-result" approach is $O(|T| * \tau)$.

4.2 Database Minimization

For enterprise database applications, it is likely that $\mathcal{D}_I$ is huge, and therefore repeatedly executing $\mathcal{E}$ on this large database during the extraction process may take an impractically long time. To tackle this issue, before embarking on the SPJ extraction, we attempt 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.

Here, the notion of empty result covers both the case where there are zero result rows, as well as when there is a single row with all null values (a corner case that can occur, for instance, with an aggregate query on an empty database) – i.e. both syntactic and semantic emptiness.

With the above definition of D_{min}, we can state the following strong observation for EQC^{-H} (EQC without HAVING):

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

Proof. Firstly, since the final result is known to be populated, the intermediate result obtained after the evaluation of the SPJ core of the query is also guaranteed to be non-empty. This is because the subsequent GAOL elements only perform computations on the intermediate result but do not add to it. Now, if we consider the *provenance* for each row r_i in the intermediate result, there will be exactly *one row* as input from each table in T_E because: (i) if there is no row from table t, r_i cannot be derived because the inner equi-join (as assumed for the query class EQC) with table t will result in an empty result; (ii) if there are $k : (k > 1)$ rows from t, $(k - 1)$ rows either do not satisfy one or more join/filter predicates and can therefore be removed from the input, or they will produce a result of more than one row since there is only a single instance of t in the query. In essence, a single-row R can be traced back to a single-row per table in D_{min}. □

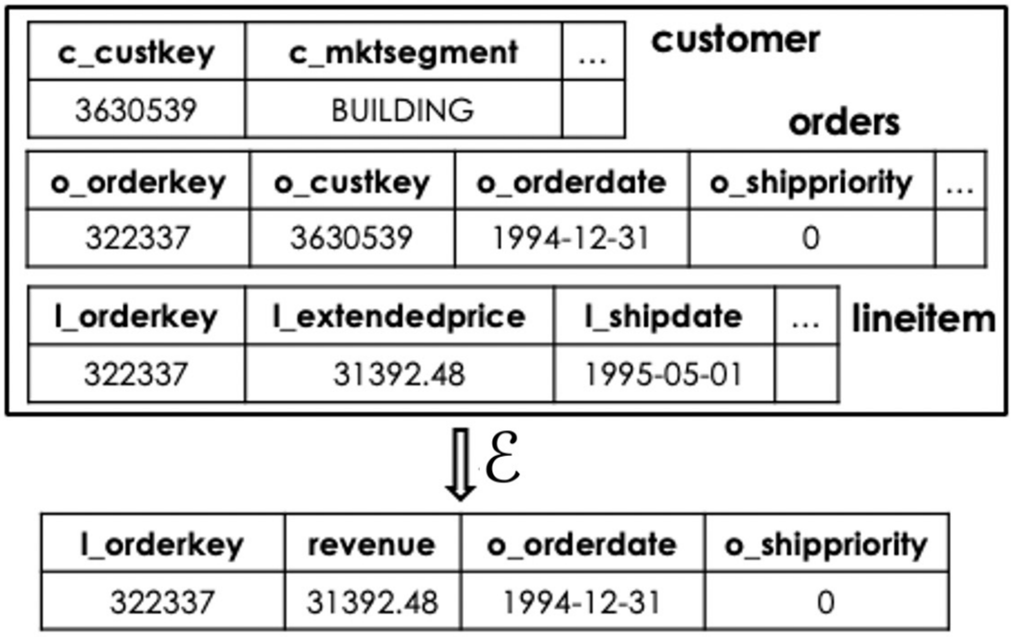
We hereafter refer to this single-row D_{min} as D^1– the reduction process to identify this database is explained next.

4.2.1 Reducing $\mathcal{D}_I$ to D^1

At first glance, it might appear trivial to identify a D^1– simply pick any row from the R obtained on $\mathcal{D}_I$ and compute its provenance using the well-established techniques in the literature (e.g. [12]) – the identified source rows from T_E constitute the single-row D^1. However, these tuple provenance techniques in the literature are predicated on *prior knowledge* of the query. This makes them unviable for identifying D^1 in our case where the query is hidden. Therefore, we implement the following iterative-reduction process instead: 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 first half. Otherwise, retain only the second half, which must, by definition, have at least one result-generating row (due to Lemma 3.1). 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.

To make the above concrete, a sample D^1 for Q3 (created from an initial 100 GB instance) is shown in Figure 6.



c_custkey	c_mktsegment	...
3630539	BUILDING	

o_orderkey	o_custkey	o_orderdate	o_shippriority	...
322337	3630539	1994-12-31	0	

l_orderkey	l_extendedprice	l_shipdate	...
322337	31392.48	1995-05-01	

l_orderkey	revenue	o_orderdate	o_shippriority
322337	31392.48	1994-12-31	0

Figure 6: D^1 for Q3

4.2.2 Time Complexity

In each iteration, the Minimizer reduces the maximum sized table to half its size and after that the application is executed. Thus, in each iteration, the query cost (as per the model in Section 3.2) is reduced by half. Therefore, the time taken by the Minimizer to reduce the database to one row can be computed as $(\frac{\tau}{2} + \frac{\tau}{4} + ... + 1)$ which is upper bounded by $O(\tau)$.

4.2.3 Sampling-based Preprocessing

The efficiency of the reduction strategy could be improved by leveraging the *sampling* methods that are natively available in most database systems. Specifically, instead of executing the MINIMIZER directly on $\mathcal{D}_I$, we could 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.

An important point to note here is that after the above-mentioned database size reduction, the remaining modules of Mutation Pipeline run $\mathcal{E}$ on D_{min} which is just a one row database. Also, the synthetic databases used in Generation Pipeline are of very small sizes (usually a handful number of rows in each table). Thus, for subsequent modules, the time for a single invocation of the executable is taken as constant.

4.3 Equi-Join Predicates

To extract the key-based equi-join predicates J_E of $\mathcal{Q}_H$, we start with SG, the original schema graph of the database. 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 attributes in the schema. (In the case of composite keys, an edge is drawn from each element of the key 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 now individually check for the presence of each CJG_E cycle in the query, using the iterative procedure shown in Algorithm 1, retaining in JG_E only those which pass the test. The check is done in the following three steps:

1. Using the CUT subroutine, a pair of edges, e_1 and e_2, is removed from a random 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_1 and CYC_2 are included as fresh candidates in CJG_E.

As a limiting case, if a cycle becomes 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 *transitivity* 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}**.

4.3.1 Proof of Correctness

Lemma 2: For a hidden query $\mathcal{Q}_H \in$ EQC, UNMASQUE correctly extracts JG_E, or equivalently, J_E.

Proof. It is trivial to see that when there is only one edge in the cycle, it will be correctly extracted as the output after removing this edge will be empty iff this edge is present in the join graph. For the edges that belong to bigger cycles, we prove the claim by contradiction. Consider an edge (u, v) that belongs to JG_E but is not extracted (i.e. a false negative). This implies that when the edge (u, v) is removed by value negation (with any other edge) the result continues to be populated. This is not possible if $(u, v) \in JG_E$ as one of the nodes from u and v is negated.

On the other hand, consider an edge $(u, v) \in C$ that is not part of JG_E but is extracted (i.e. a false positive). This implies that when the edge (u, v) is explicitly removed along with any other edge (x, y) by value negation, the result becomes empty. As there is no other filter on key attributes and $(u, v) \notin JG_E$, every other edge in C must belong to the join graph. Now due to inner equi-join assumption, (u, v) also belongs to the join graph as it can be inferred from the other edges of cycle C, a contradiction. □

Algorithm 1: Extracting Equi-Join Graph JG_E

```
$CJG_E \leftarrow$ Candidate Cycles, $JG_E \leftarrow \phi$
while There is at least one cycle in $CJG_E$ do
    $CYC \leftarrow$ Any candidate cycle from $CJG_E$
    if $CYC$ contains a single edge $(v_1, v_2)$ then
        $D^1_{mut} \leftarrow$ Negate($D^1$, $\{v_1\}$)
        If $\mathcal{E}$ ($D^1_{mut}) = \phi$ then $JG_E \leftarrow JG_E \cup CYC$
        $CJG_E \leftarrow CJG_E$ / $CYC$
    else
        foreach pair of edges $(e_1, e_2) \in CYC$ do
            $CYC_1, CYC_2 = Cut(CYC, e_1, e_2)$
            $D^1_{mut} \leftarrow$ Negate($D^1$, $CYC_1$)
            if $\mathcal{E}$ ($D^1_{mut}) = \phi$ then
                Add $e_1$ and $e_2$ back to $CYC$
            else
                $CJG_E \leftarrow (CJG_E - CYC) \cup CYC_1 \cup CYC_2$
                break    //Go to the start of while loop
            end
        end
        $JG_E \leftarrow JG_E \cup CYC$;  $CJG_E \leftarrow CJG_E$ / $CYC$
    end
end
```

4.3.2 Time Complexity

Let E denote the set of edges in SG and C^{key} denote the set of key attributes taken from the tables in T_E. Then, finding connected components of the graph can be performed in $O(C^{key} + E)$ and converting each component into the respective cycle can be done in $O(C^{key})$ in the worst case as it is equivalent to creating a cycle from the nodes in the component. After that, checking presence of each edge requires at most $\binom{C^{key}}{2}$ iterations. Thus overall time complexity for join predicate extraction comes out to be $O(E * |C^{key}|^2)$.

Case	$R_1 = \phi$	$R_2 = \phi$	**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

Table 2: Filter Predicate Cases

4.4 Filter Predicates (non-key)

We start by assuming that all columns in C_E are potential candidates for the filter predicates F_E in $\mathcal{Q}_H$ (as per EQC^{-H}, each such attribute can appear in at most one filter predicate). Each of them is then checked in turn with the following procedure: First, we evaluate whether there is a nullity predicate on the column. If an *IS NULL* predicate is not present, we investigate whether there is an arithmetic predicate, and if yes, the filter value(s) for the predicate are identified.

It is relatively easy to check for nullity predicates and, more generally, predicates on any data types with small finite domains (e.g. Boolean), by simply mutating the attribute with each possible value in its domain and observing the result – empty or populated – of running $\mathcal{E}$ on these mutations. The procedure for general numeric and textual attributes is, however, more involved, as explained below.

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 range 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 generic format – for example, $A < 25$ can be written as $i_{min} \leq A \leq 24$.

To check for presence of a filter predicate on column A, we first create a 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. We get another result – call it R_2 – by applying the same process with i_{max}. Now, the existence of a filter predicate is determined based on one of the four disjoint cases shown in Table 2.

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

We apply the above procedure for each of the non-key columns that feature in T_E. Since the value of only one column (say $t.A$) is changed at a time, it ensures that any change in the result is solely due to the change in $t.A$. This enumerative method ensures that we correctly identify filter predicates of the type *column op value* with *op* $\in \{=, <, >, \leq, \geq, between\}$ for each numeric database column.

We can easily extend the integer approach to *float* data types with *fixed precision*, by first identifying the *integral* bounds with the above procedure and then executing a second binary search to identify the *fractional* bounds. For example, with l_i and r_i as the integral bounds identified in the first step, and assuming a precision of 2, we search l in $((l_i - 1).00, l_i.00]$ and r in $[r_i.00, r_i.99)$ in the second step.

Time Complexity Let u denote the range of the attribute's data type. Now, we require two table updates and two calls to the executable to determine one of the four cases in Table 2 which is an $O(1)$ operation. After that, if the attribute has a filter, we require $\log u$ table updates and corresponding executable calls. Thus, the total time complexity of filter predicate extraction for an attribute is $O(\log u)$.

4.4.2 Date Columns

Extracting predicates on *date* columns is identical to that of integers, with the minimum and maximum expressible dates in the database engine serving as the initial range, and days as the difference unit. For example, after identifying filter of type $A \leq r$ on o_orderdate, we apply binary search strategy in range [*'1994-12-31', r*] (assuming '1994-12-31' is the value of o_orderdate in D^1) and r is the greatest allowed date value in the database engine (for PostgreSQL, $r = 5874897AD$). Note that the same strategy can be applied to other *datetime* type columns with the corresponding change in the resolution of values.

4.4.3 Boolean Columns

With a single row, a boolean column can have only one of True or False values. Therefore, to identify a filter on boolean column $t.A$, we create a D^1_{mut} by replacing its value in D^1 with True (resp. False) if the current value in D^1 is False (resp. True) and get the result. If the result is empty, add *"A* = False*"* (resp. *"A* = True*"*) to F_E.

4.4.4 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, we create two different D^1_{mut} instances by replacing the value of A initially with an empty string and then with a single character string – say *"a"*. $\mathcal{E}$ is invoked on both these instances, and we conclude that a filter predicate is in operation iff the result is empty in one or both 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. For 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.

Upon confirming the existence of a filter predicate on A, we extract the specific predicate in two steps. Before getting into the details, we define a term called *Minimal Qualifying String (MQS)*. Given a character/string expression *val*, its MQS is the string obtained by removing all occurrences of '%' from *val*. For example, "UP_" is the MQS for *"%UP_%"*. Note that each character of MQS, with the exception of wildcard '_', must be present in the data string to satisfy the filter predicate. 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 formal procedure to identify *MQS* is detailed in Algorithm 2. The basic idea here is to loop through all the characters of *rep_str* and determine whether it is present as an intrinsic character of the MQS or invoked through the wildcards ('_' or '%'). This distinction is achieved by replacing, in turn, each character of *rep_str* in D^1 with some other character, executing $\mathcal{E}$ on this mutated database, and checking whether the result is empty – if yes, the replaced character is part of MQS; if no, this character was invoked through wildcards. In this case, further action is taken to identify the correct wildcard character. Note that in case the character in *rep_str* occurs more than once without any intrinsic character in between, and only one of them is part of MQS, our procedure puts the rightmost character in MQS.

Lemma 3: For a query in EQC, Algorithm 2 correctly identifies MQS for a filter predicate on character attribute.

Algorithm 2: Identifying *MQS*

```
Input: Column $A$, rep_str, $D^1$
$itr = 0$; $MQS =$ “”
while $itr < len(rep\_str)$ do
    $temp = rep\_str$
    $temp[itr] = c$ where $c \neq rep\_str[itr]$
    $D^1_{mut} \leftarrow D^1$ with value $temp$ in column $A$
    if $\mathcal{E}$ ($D^1_{mut}$) $= \phi$ then
        $MQS$.append($rep\_str[itr_{++}]$)
    else
        $temp$.remove_char_at($itr$)
        $D^1_{mut} \leftarrow D^1$ with value $temp$ in column $A$
        if $\mathcal{E}$ ($D^1_{mut}$) $= \phi$ then
            $MQS$.append('_'); $itr_{++}$
        else
            $rep\_str$.remove_char_at($itr$)
        end
    end
end
```

Proof. The correctness of the algorithm 2 can be established using contradiction for each of the possible failed cases. For example, let us say a character ‘a’ belonged to *MQS* but the procedure fails to identify it. This means that after removing ‘a’ from *rep_str*, the result is still non-empty (the filter condition was satisfied). This is possible when ‘a’ occurs more than once in *rep_str* and there is at least one occurrence which is part of the replacement for wildcard ‘%’. However, the procedure will keep removing ‘a’ until there is no occurrence left which is part of replacement for wildcard ‘%’. After that, removing ‘a’ will lead the corresponding filter predicate to fail. If this is not the case, ‘a’ is not present in the *MQS*, a contradiction. Similarly, the correctness for other cases can be proved. □

After obtaining the MQS, we need to find the locations (if any) in the string where ‘%’ is to be placed to get the actual filter value. This is achieved with the following simple linear procedure: For each pair of consecutive characters in MQS, we insert a random character that is different from both these characters and replace the current value in column A with this new string. A populated result for $\mathcal{E}$ on this mutated database instance indicates the existence of ‘%’ between the two characters. The inserted character is removed after each iteration and we start with the initial MQS for each successive pair of consecutive characters. This makes sure that we correctly identify the locations of ‘%’ without exceeding the character length limit for A. In the specific case of $Q3$, the predicate value for c_mktsegment turns out to be the MQS itself, namely *‘BUILDING’*.

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

F_E **= { o_orderdate $\leq$ date ‘1995-03-14’ ,**
l_shipdate $\geq$ date ‘1995-03-16’ ,
c_mktsegment = ‘BUILDING’ }

Time Complexity Let len be the character limit of the textual attribute’s data-type. For example, *varchar(50)* implies $len = 50$. Then, the time complexity for Algorithm 2 is $O(len)$ as a single pass through rep_str is performed using the while loop in the algorithm and each iteration performs constant time operations. After that, finding places for '%' requires a single pass through MQS. Thus, time complexity for textual predicate extraction is linear in the maximum number of characters allowed by the corresponding data-type.

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

The identification of projections is rendered tricky since they may appear in a variety of different forms – native columns, renamed columns, aggregation functions on the columns, or UDFs with column variables. 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 more columns is discussed at last.

Let O denote the 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:

$$aA + bB + cAB + d = 0 \tag{1}$$

where a, b, c, d are constant coefficients. With this framework, the extraction process proceeds, as explained below, in two steps: (i) Dependency List Identification, which identifies the identities of A, B, and (ii) Function Identification, which identifies 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.

However, a subtle point here is that even in the simplified two-variable scenario, a single pass through all the database columns may not always be sufficient to obtain the complete dependency list of O. For instance, if the value of column A in D^1 coincidentally happens to be $\frac{-b}{c}$, the entry in column B has no impact on O, irrespective of its value. We say that A is a blocking column and B is the blocked column for that database instance. Similarly, if the B value happens to be $\frac{-a}{c}$, column A is rendered ineffectual. To address such boundary conditions, we perform a second iteration if the dependency list has less than two columns after the first. Prior to the second iteration, the values in all the database columns are changed to new s-values, thereby ensuring that both attribute dependencies, if present, become visible.

Finally, as a special case, note that if the output column represents *count(*)*, its dependency list will be empty.

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. Since we have a non-homogeneous equation in 4 unknowns, it can be easily solved by creating 4 different D^1_{mut} instances such that the resultant equations are linearly independent. This is achieved by randomly mutating the values of A and B, checking whether the new vector $[A, B, AB, 1]$ is linearly independent from the vectors generated so far, and stopping when four such vectors have been found. With regard to Q3, the revenue output column depends on A = l_extendedprice and B = l_discount. The sample four equations, corresponding to output column revenue, generated in our experiments are as below:

$$1.a + 2.b + 2.c + d = -1 \tag{2}$$

$$2.a + 1.b + 2.c + d = 0 \tag{3}$$

$$2.a + 3.b + 6.c + d = -4 \tag{4}$$

$$1.a + 4.b + 4.c + d = -3 \tag{5}$$

Solving the above system results in coefficient values: $a = 1, b = 0, c = -1, d = 0$, producing the function seen in Q3. For the remaining output columns, which are all dependent on only a single database column, we get the function of the form $aA + d$ with $a = 1, d = 0$ – i.e. a native column.

Thus for query Q3, we obtain 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 we show the above set 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 did not have to concern ourselves with these aggregation functions in the current stage because our extraction techniques operated on *single-row* databases, in which case all aggregation functions are identical with regard to their values.

4.5.3 Proof of Correctness

Lemma 4: The above procedure correctly identifies scalar functions of up to two variables in P_E.

Proof. Firstly, we correctly identify the dependency list for an output column because the second iteration with different values in database columns ensures that no column is left out due to the blocking condition mentioned above. Also, it can be seen that any column which is not part of the function, will not be added in the dependency list because we perform change only one column at a time ensuring that the change in the output column is due to the change in that particular database column only. After identifying the dependency list, if the function satisfies the linearity condition mentioned in equation 1, it will be correctly identified as we ensure to generate a system of equation with a unique solution (by checking that vectors $[A, B, AB, 1]$ generated from different values of A and B are linearly independent). □

4.5.4 Time Complexity

To get the dependence of a database column in an output function, we change its value and observe the effect in all the output columns of the result. As mentioned above, for a two variable (database columns) output function, two passes through C_E (the columns belonging to the tables in T_E) are enough. After that, we generate four equations and solve them using standard techniques (e.g. Gauss Elimination Method) which, when the number of variables is small, represent constant time operations. Thus time complexity for projection identification (assuming dependency list size up to 2 for each function) is $O(C_E)$.

Generalization Note that the above process can be generalized to m column variables in the function if we are able to generate 2^m different D^1_{mut} instances. In such a case, the time complexity becomes heavily dependent on the function identification process. In general, for an m variable function, we need to solve a system of 2^m equations which is a $O((2^m)^3)$ operation.

4.5.5 CASE Statements

We can handle the CASE switch statements of the type (CASE WHEN $column\ op\ value$ THEN $const1$ ELSE $const2$ END) on categorical domains, such as those seen in TPC-H Q12. If no numeric function is found and the column is of categorical domain, its values are altered to created D^1_{mut} instances and the output is mapped to the corresponding column values. These mapping are then converted to the corresponding case statements. For example, let us say the output column takes value 0 when a particular attribute named “gender" takes value ‘M’ and its 1 otherwise. It is transformed to the following case statement: (CASE WHEN gender = ‘M’ THEN 0 ELSE 1 END).

4.5.6 Ancillary Functions

Finally, ancillary functions such as *substring, casting, etc.* can also be extracted by mapping the input-output values and then checking for these specific functions.

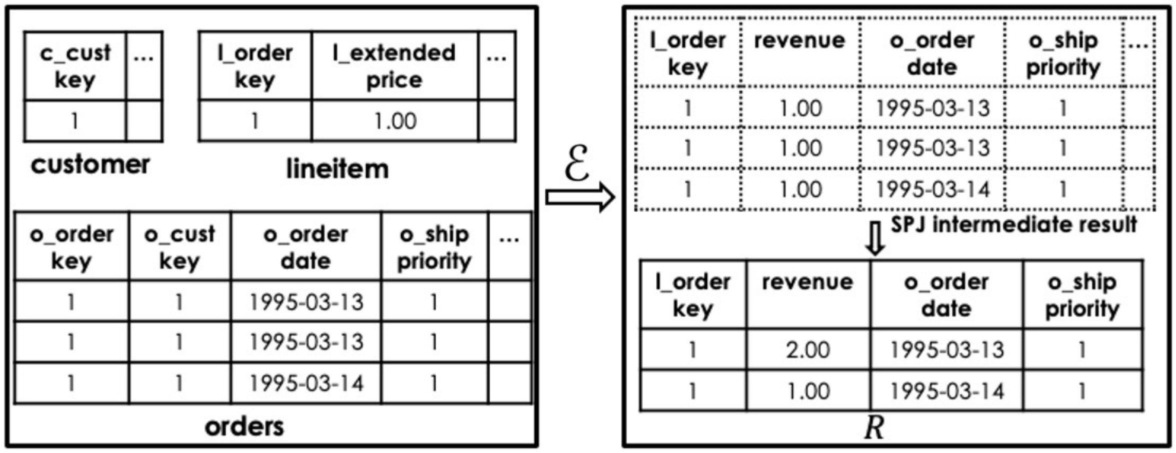
5 Generation Pipeline

The GAOL part of the query, corresponding to the GROUP BY (G_E), AGGREGATION (A_E), ORDER BY ($\overrightarrow{\mathcal{O}_E}$) and LIMIT ($l_E$) clauses, is extracted in the Generation Pipeline segment of UNMASQUE. Here, synthetically generated miniscule databases are used for all the extractions, as described in the remainder of this section.

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 previously in the Mutation Pipeline) since their presence or absence in G_E does not impact the query result.

Assume for the moment that we have constructed a D_{gen} such that the (invisible) *intermediate* result produced by the SPJ core of $\mathcal{Q}_H$ 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. We call this as desired “target intermediate result" for column $t.A$. 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 the techniques presented in [19, 26] for identifying query mutants.



customer

c_cust key	...
1	

lineitem

l_order key	l_extended price	...
1	1.00	

orders

o_order key	o_cust key	o_order date	o_ship priority	...
1	1	1995-03-13	1	
1	1	1995-03-13	1	
1	1	1995-03-14	1	

l_order key	revenue	o_order date	o_ship priority	...
1	1.00	1995-03-13	1	
1	1.00	1995-03-13	1	
1	1.00	1995-03-14	1	

R

l_order key	revenue	o_order date	o_ship priority
1	2.00	1995-03-13	1
1	1.00	1995-03-14	1

Figure 7: D_{gen} for Grouping on o_orderdate (Q3)

5.1.1 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 $r = 1$ (consistent with the EQC^{-H} assumption that 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.

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

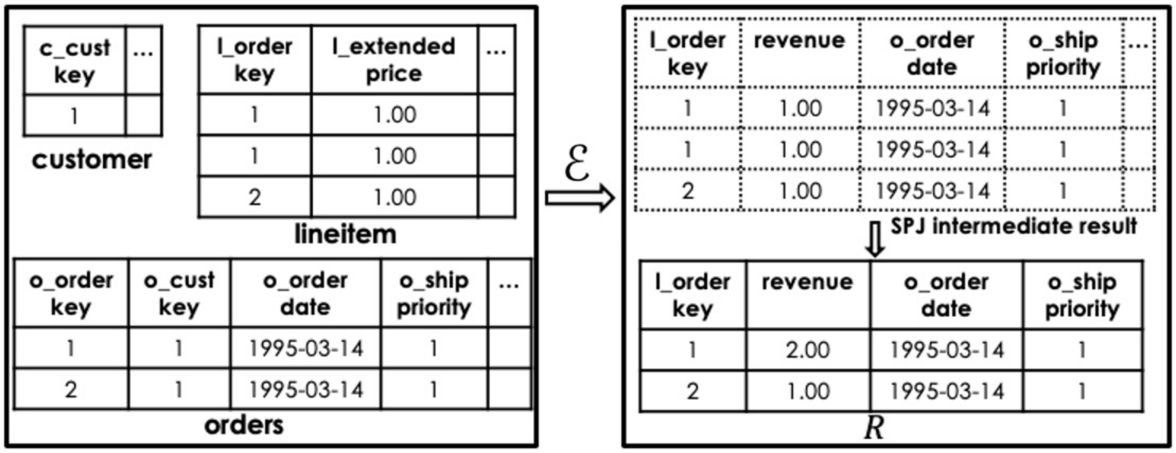
An example D_{gen} for checking the presence of o_orderdate in the G_E of **Q3** is shown in Figure 7. Here, the ORDERS table features 3 rows with p = '1995-03-13' and q = '1995-03-14', while the remaining tables, LINEITEM and CUSTOMER, have a single row apiece.

Case 2: $t.A \in JG_E$ Here, we generate 3 rows for table t, 2 rows for all tables t' having a column $t'.B$ such that there is 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 8. 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 structure for the intermediate SPJ result. Namely, it contains exactly 3 rows with all columns having the same value in all of them, except for the attribute under test which has *two* values across these rows.

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



customer

c_cust key	...
1	

lineitem

l_order key	l_extended price	...
1	1.00	
1	1.00	
2	1.00	

orders

o_order key	o_cust key	o_order date	o_ship priority	...
1	1	1995-03-14	1	
2	1	1995-03-14	1	

SPJ intermediate result

l_order key	revenue	o_order date	o_ship priority	...
1	1.00	1995-03-14	1	
1	1.00	1995-03-14	1	
2	1.00	1995-03-14	1	

R

l_order key	revenue	o_order date	o_ship priority
1	2.00	1995-03-14	1
2	1.00	1995-03-14	1

Figure 8: D_{gen} for Grouping on l_orderkey (Q3)

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}**.

5.1.2 Proof of Correctness

Lemma 5: The above procedure identifies the columns in G_E correctly.

Proof. If we prove that for each attribute, the above data generation technique ensures that we get the "target intermediate result" for the queries in EQC^{-H}, then the presence of that attribute can be identified correctly. This statement can be proved by contradiction. Let us suppose an attribute $t.A \notin JG_E$ belongs to G_E, but our procedure failed to identify it. It implies that the desired "target intermediate result" was not generated. However, following the construction mechanism for **Case 1** above, we should have gotten the "target intermediate result" for a query in EQC^{-H}. It implies that $\mathcal{Q}_H \notin EQC^{-H}$. Similarly by taking contradictory cases and following the construction above, the correctness for other procedures can be proved. □

5.1.3 Time Complexity

As discussed previously in Section 4.2, the synthetic databases are of very small sizes. So the time for a single executable invocation is taken as constant. Thus the time complexity of G_E identification can be calculated by the number of total executable invocations and the effort involved in generating the database for that invocation. For each column in C_E (set of columns in the T_E tables), we make a pass to assign values as per the data generation technique defined above, and make one executable invocation on the resultant database. Thus, the overall time complexity of G_E extraction is $O({C_E}^2)$.

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 the end of the section.

As described in Section 4.5, the Projection Extractor models each output column O as a function of the database columns within its dependency list. With this framework, 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 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 $o_1 \neq 0$, $o_1 \neq o_2$. Note that the two arguments differ only in s_i and s_i'. 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. Further, that k satisfies the following *forbidden-value* constraint:

$$k \notin \left\{0, o_1 - 1, o_2 - 1, \frac{o_1 - o_2}{o_1}, \frac{1 - o_2}{o_1 - 1}, \frac{(o_2 - 2) \pm \sqrt{(o_1 - 2)^2 - 4(1 - o_2)}}{2}\right\} \tag{6}$$

	Max = o_2	Sum = $ko_1 + o_2$	Avg = $(ko_1 + o_2)/(k+1)$	Count = k + 1
Min = o_1	X	$k \neq (o_1 - o_2)/o_1$	X	$k \neq (o_1 - 1)$
	Max = o_2	$k \neq 0$	X	$k \neq (o_2 - 1)$
		Sum = $ko_1 + o_2$	$k \neq 0$	$k \neq (1 - o_2)/(o_1 - 1)$ *if* $o_1 \neq 1$
			Avg = $(ko_1 + o_2)/(k+1)$	$k \neq \frac{(o_1 - 2) \pm \sqrt{(o_1 - 2)^2 - 4(1 - o_2)}}{2}$

X denotes no condition

Figure 9: Constraints on k

These constraints on k have been derived by computing *pairwise equivalences* of the five aggregation functions, and forbidding all the k values that result in any equality across functions. The table shown in Figure 9 gives individual constraint on the value of k which results from *pairwise equivalences*. For example, the constraint $k \neq (o_1 - 1)$ is required for *count* and *min* aggregates to result in different values. Now, additionally if we ensure that the G_E attributes are assigned common values in all the rows, the result of $\mathcal{E}$ will be the target result.

5.2.1 Generating D_{gen}

First, we choose the i^{th} 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 6. 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 $t.A_i$ being assigned value s_i in k rows and value 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 10. 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()*.

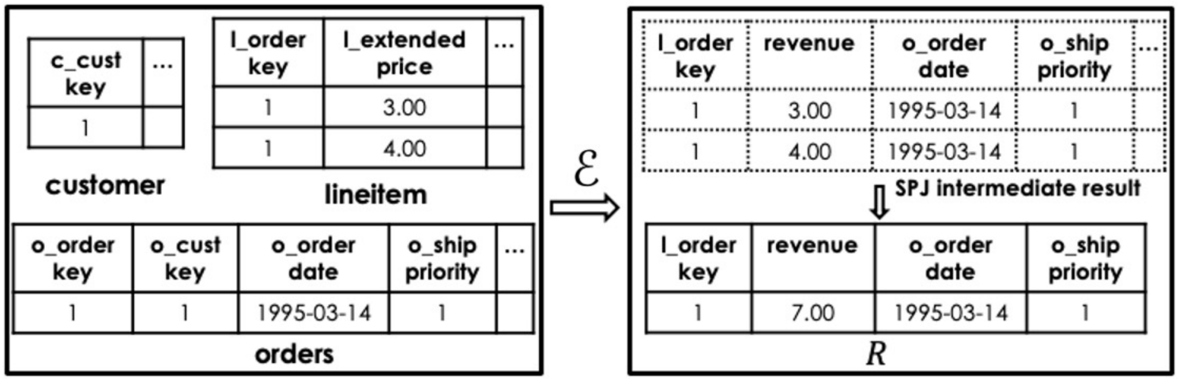


Figure 10: D_{gen} for Aggregation on revenue UDF (**Q3**)

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}**

5.2.2 Proof of Correctness

Lemma 6: The above procedure identifies the columns in A_E correctly.

Proof. Let us say that for each column, we are able to generate the database such that the result of $\mathcal{Q}_H$ is the "target result". Then it is clear that the correct aggregation will be identified. Now, the reason that the target result is produced is (i) the result 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 output of all the aggregations for O. (As a special case, if f_o is a constant function or a function of only the columns in G_E, we are forced to have $a_i = a'_i$ and hence, $o_1 = o_2 = c$. Here, the k constraint reduces to $k \notin \{0, c - 1\}$ and since multiple aggregations on f_o are equivalent (e.g. *min(), max(), avg()*), any of them can be taken as the final choice.) $\square$

5.2.3 Time Complexity

The time complexity of A_E extraction is similar to that for G_E extraction with the only change being that in each executable invocation, one of the database tables will contain k rows. However, as k is taken as the smallest value satisfying equation 6 which restricts k to be not equal to 6 different values at maximum. Thus, there always exists a k such that k satisfies equation 6 and $k < 7$, which is a very small number of rows. Hence, here also, the time for one single executable invocation can be taken as constant. Thus, the time complexity for A_E extraction is the same as for G_E extraction, i.e. $O({C_E}^2)$.

5.2.4 Extension to *DISTINCT* keyword

We now consider the case where the aggregation may be present with the *DISTINCT* keyword. Here, we first invoke the above method which will result in one of the following three cases:

Case1: No aggregation is identified: The outcomes of *min()* and *max()* aggregations do not depend on *DISTINCT*. Therefore, in this particular case, the aggregation on f_o can be one of *sum(DISTINCT f_o), avg(DISTINCT f_o)* or *count(DISTINCT f_o)*. To identify the correct aggregation, we add $(o_1 + o_2) \notin \{2, 4\}$ as an extra condition on o_1 and o_2 while generating the s-value arguments. This ensures that the three candidate aggregations result in distinct computed values.

Case2: min() or max() aggregation is identified: No action is required as *min()* or *max()* produce exactly the same result with or without *DISTINCT*.

Case3: Aggregation other than min() or max() is identified: Here, the possible aggregations on f_o are *sum(DISTINCT f_o), avg(DISTINCT f_o), count(DISTINCT f_o)*, or the one identified without distinct. In such a case, we generate databases to prune this list one by one. For example, let *sum (f_o)* be the identified projection. To prune one of *sum(f_o)* and *sum(DISTINCT f_o)*, we generate a D_{gen} instance with $k = 2$ and $o_1 \neq 0$. Similarly, other candidates can be pruned as well. As a final note, in the case of equivalent aggregations, any among them can be chosen.

5.2.5 Extension to Non-numeric Columns

For a non-numeric column A, the *sum()* or *avg()* aggregations do not apply. The existence of *count()* aggregation can be identified in a manner analogous to *count()* aggregations for numeric columns. Finally, we need to check the existence of *min()* or *max()* aggregations. In such a case, we take $k = 1$ and identify two different values a and b from the domain of A such that the corresponding output column function returns two different values. Based on the outcome, *min()* or *max()* aggregation can be identified.

5.3 Order By

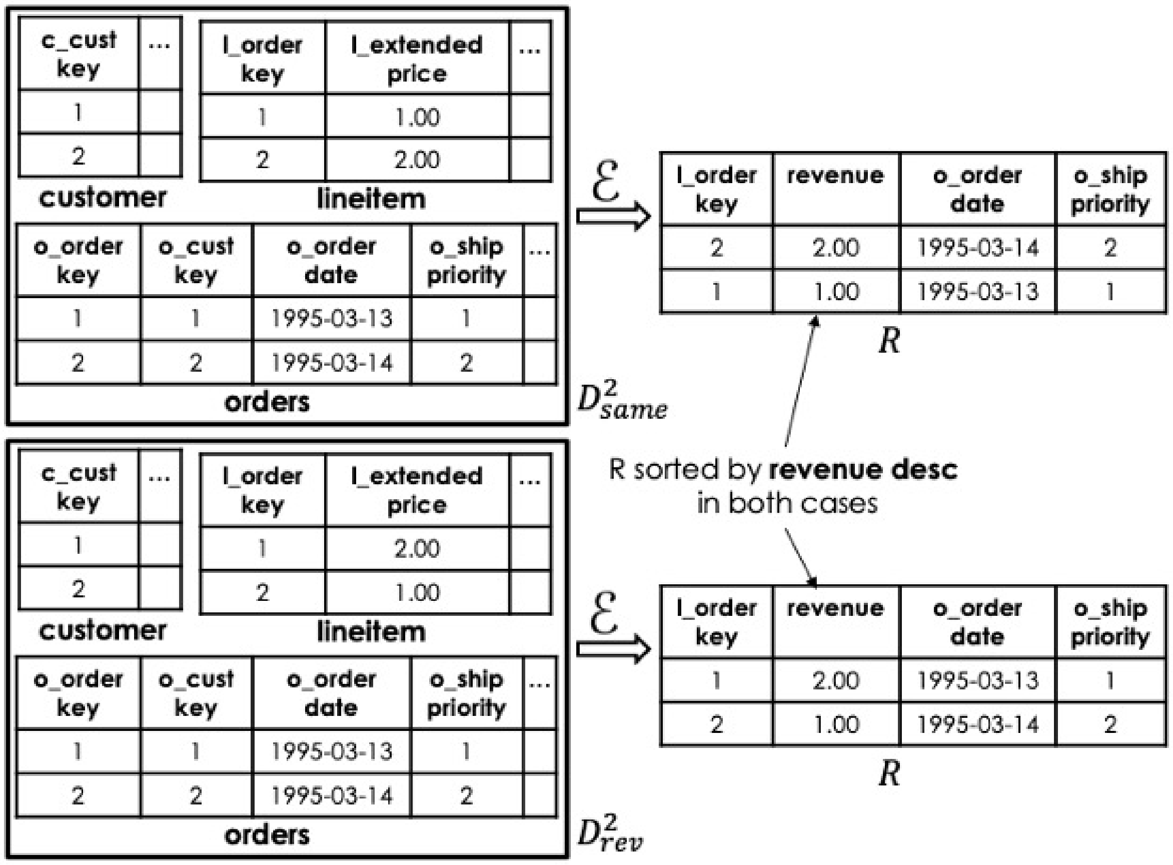
We now move on to identifying the sequence of columns present in $\overrightarrow{\mathcal{O}_E}$. A basic difficulty here is that the result of a query can be in a particular order either due to: (i) explicit ORDER BY clause in the query or (ii) a particular plan choice (e.g. Index-based access or Sort-Merge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

Here, we expect that each database column occurs in the dependency list of at most one output column. Further, for simplicity, we assume that $count() \notin A_E$ and that no aggregated output column is a constant function – the procedure to handle these special cases is described at last.

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 $\overrightarrow{\mathcal{O}_E}$ 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 $\overrightarrow{\mathcal{O}_E}$, or (ii) no sort order can be identified for the current index position.

To check for the existence of an output column O in $\overrightarrow{\mathcal{O}_E}$, 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 11.



D^2_{same}:

customer

c_cust key	...
1	
2	

lineitem

l_order key	l_extended price	...
1	1.00	
2	2.00	

orders

o_order key	o_cust key	o_order date	o_ship priority	...
1	1	1995-03-13	1	
2	2	1995-03-14	2	

R

l_order key	revenue	o_order date	o_ship priority
2	2.00	1995-03-14	2
1	1.00	1995-03-13	1

D^2_{rev}:

customer

c_cust key	...
1	
2	

lineitem

l_order key	l_extended price	...
1	2.00	
2	1.00	

orders

o_order key	o_cust key	o_order date	o_ship priority	...
1	1	1995-03-13	1	
2	2	1995-03-14	2	

R

l_order key	revenue	o_order date	o_ship priority
1	2.00	1995-03-13	1
2	1.00	1995-03-14	2

Figure 11: 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, namely (i) S_1, the set of output columns already present in $\overrightarrow{\mathcal{O}_E}$ (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 identified in Section 4.5 for output column O. 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. On the other hand, 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 separately in extensions part). 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 $\overrightarrow{\mathcal{O}_E}$ 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:

$\overrightarrow{\mathcal{O}_E}$ **= {revenue desc, o_orderdate asc}**

5.3.2 Proof of Correctness

Lemma 7: With the above procedure, if O is not the rightful column at position i in $\overrightarrow{\mathcal{O}_E}$, and another column O' is actually the correct choice, then the values in O will not be sorted in the same order in the two results.

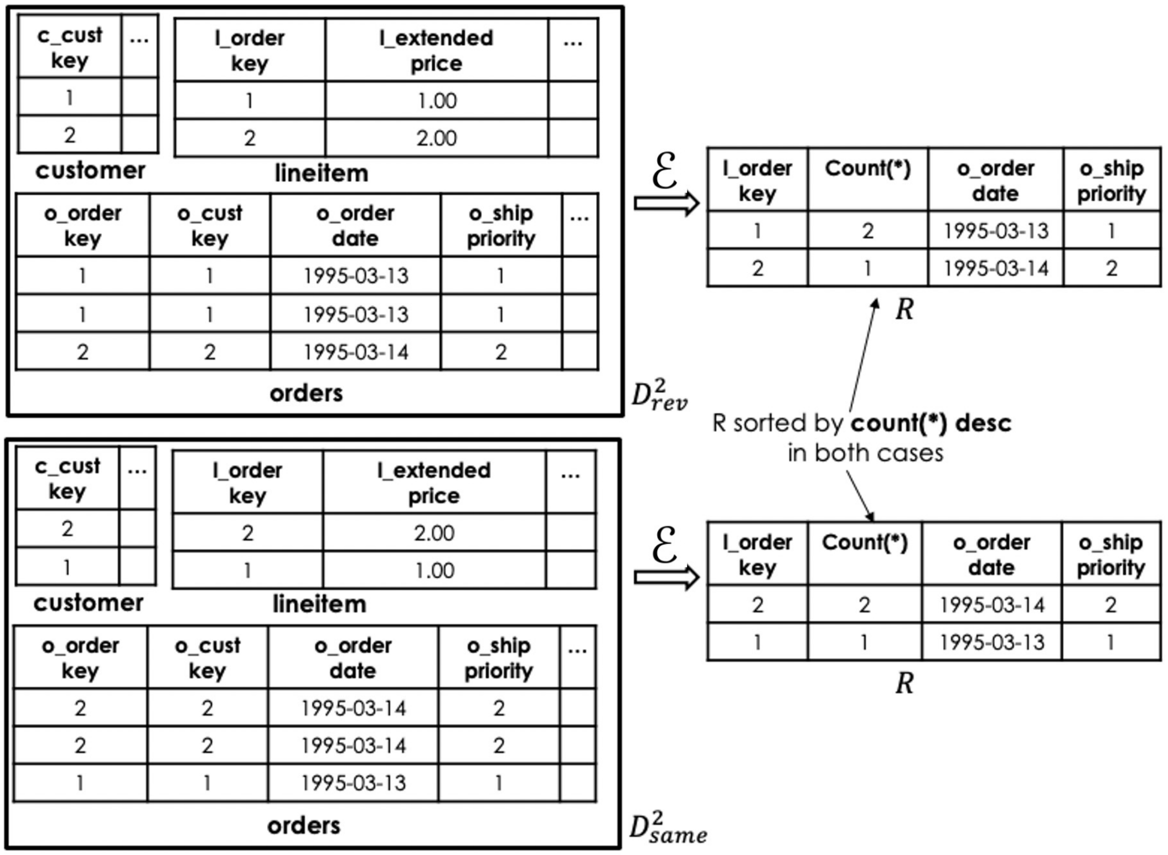
Proof. Firstly, as each column in the existing identified $\overrightarrow{\mathcal{O}_E}$ is assigned the same value in both the rows, they have no effect on the ordering induced by other attributes. Now, let us say that the next attribute in $\overrightarrow{\mathcal{O}_E}$ is O' (asc) but UNMASQUE extracts O. In that case, the result corresponding to D^2_{same} will contain the values in O sorted in the ascending order. But in the result corresponding to D^2_{rev}, the values in O will be sorted in descending order (due to ascending order on O'), a contradiction. □

5.3.3 Time Complexity

For each column in C_E (set of columns in the T_E tables), we once make a pass to assign values for D^2_{same} and then D^2_{rev} as per the data generation technique defined above, and make two executable invocation on the resultant databases (one for D^2_{same} and one for D^2_{rev}). Creating the sets S_1, S_2 and S_3 requires one pass through all the columns and classifying each column in one of the above sets which is an $O(C_E\ )$ operation. Thus, the overall time complexity of $\overrightarrow{\mathcal{O}_E}$ extraction is $O({C_E}^{\ 2})$.

5.3.4 Extension 1: *count(*)* $\in A_E$

In the case when *count(*)* $\in A_E$, having two rows in each of the tables is not enough as the *count()* value for both the groups will be one. Therefore, we need an intermediate result (on which grouping will be applied) with three rows such that two rows form one group and the third row forms another group.



c_custkey	...
1	
2	

l_orderkey	l_extendedprice	...
1	1.00	
2	2.00	

o_orderkey	o_custkey	o_orderdate	o_shippriority	...
1	1	1995-03-13	1	
1	1	1995-03-13	1	
2	2	1995-03-14	2	

l_orderkey	Count(*)	o_orderdate	o_shippriority
1	2	1995-03-13	1
2	1	1995-03-14	2

c_custkey	...
2	
1	

l_orderkey	l_extendedprice	...
2	2.00	
1	1.00	

o_orderkey	o_custkey	o_orderdate	o_shippriority	...
2	2	1995-03-14	2	
2	2	1995-03-14	2	
1	1	1995-03-13	1	

l_orderkey	Count(*)	o_orderdate	o_shippriority
2	2	1995-03-14	2
1	1	1995-03-13	1

Figure 12: D^2_{same} and D^2_{rev} for Ordering on count(*) (Hypothetical scenario:Q3)

Also, the values in the rows should be according to the order desired after grouping of the intermediate result. So the data generation process is as follows:

To generate data for D^2_{rev}, we first choose a table t with at least one attribute in G_E that can take two different values and is not present as an argument to any column in S_1. For each output column function $f_o \in S_1$, we take argument value $(a_1, .., a_n)$ and assign same values in both the rows to corresponding columns in the table. For each output column function $f_o \notin S_1$, we take two different argument values $(a_1, .., a_n)$ and $(b_1, ..b_n)$ and assign values to corresponding columns in the table. In case the column is a key column, we take fixed values 1 and 2. For all the other columns of other tables t', we generate two rows with each attribute having two different values (p and q) such that $p < q$. In case of key attributes, take $p = 1$ and $q = 2$. In other cases, take p and q as s-values. Note that in the above procedure, if we encounter an attribute with an equality filter predicate, we take $p = q = val$ where val is an s-value.

Data generation for D^2_{same} is similar to that of D^2_{rev} with the only change being that the values of p and q are now swapped. The further procedure of running $\mathcal{E}$ and analyzing the results is the same as that explained in Section 5.3.1. A sample D^2_{same} and D^2_{rev} database instance for a hypothetical scenario where revenue is replaced by *count(*)* is shown in Figure 12.

Lastly, for the case *count(DISTINCT* $t.A$*)* $\in A_E$, the data generation process is identical except that A is assigned values (p, q, p) in both the cases.

5.3.5 Extension 2: $t.A$: ("$t.A = val$" $\in F_E \wedge (agg\_func(t.A) \in A_E)$)

In case there is a *min()*, *max()* or *avg()* aggregation on A, the attribute can be treated as a natively projected attribute because each group in the output will have exactly the same value for A. Now, if *sum*($t.A$) $\in A_E$, the data generation process is the same as in Extension 1.

A closing note on the potential for spurious columns appearing in G_E due to plan-induced ordering: Since D^2_{same} and D^2_{rev} are extremely small in size, it is unlikely that the database engine will choose a plan with sort-based operators – for instance, it would be reasonable to expect a sequential scan rather than index access, and nested-loops join rather than sort-merge.

5.4 Limit

If the query is an SPJA query, there is no need to extract l_E since there can be only *one* row in any populated result. But in the general SPJGAOL case, the only way to extract l_E is to generate a database instance such that $\mathcal{E}$ produces more than l_E rows in the result R, subject to a maximum limit imposed by the GROUP BY clause.

The 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 $n_1 * n_2 * n_3 * ... = l_E^{max}$ groups in the result. Thus, l_E values up to l_E^{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 $r(> 1)$. We set $a = max$ (4, cardinality of R) to be consistent with our extraction requirement for G_E which required a permissible result cardinality of upto 3 rows. And r can be set to a convenient value that provides a good tradeoff between the number of iterations (which will be high with small r) and the setup cost of each iteration (which will be high with large r). In our experiments, $r = 10$ was used. This appears reasonable given that the l_E value is typically a small number in most applications – for instance, in TPC-H, the maximum is 100, and in general, we do not expect the value to be more than a few hundreds at most.

5.4.1 Generating D_{gen} for desired R cardinality

To get n rows in the result 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 the attributes appearing in JG_E are assigned values $(1, 2, 3, ..., n)$ and the other attributes are assigned any value satisfying their filter predicates (if any). If the result of applying $\mathcal{E}$ on this database contains m rows with $m < n$, then we can conclude that LIMIT is in operation and equal to m. With the above procedure, we finally obtain l_E **= 10** for Q3.

5.4.2 Time Complexity

To minimize the number of processed rows, we generate databases such the product of rows in the tables in T_E is minimal. For example, to get a rows in output, we generate a rows in one table (such that there is a group by attribute in that table which, after applying filter predicates, allows a different values) and a single row in each of the other table. Thus, to identify the limit value, we iteratively generate databases whose sizes (in terms of row-cardinality product of T_E) follow a *geometric progression* with initial value a and common ratio r (> 1) until we exceed l_E or reach the upper bound of l_E^{max}.

Using the cost model defined in Section 3.2, the time complexity to identify limit can be computed as $(a + ar + ar^2 + ... + ar^m)$ where m is a value such that $l_E \leq ar^m < r * l_E$. This sum can be upper bounded by $O(r^{\frac{\log l_E}{\log r}})$, which is $O(l_E)$.

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 through set difference, and a non-zero outcome indicates an error. Further, physical ordering equivalence is verified by computing position-dependent checksums on the results.

Second, we leverage the XData grading tool [19], 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

Having described the functioning of the UNMASQUE tool, we now move on to empirically evaluating its efficacy and its efficiency. All the experiments were hosted 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

Comparison with REGAL To begin with, we compare UNMASQUE against QRE techniques – specifically, the state-of-the-art REGAL tool [23]. 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 13, which shows the performance of UNMASQUE and REGAL on 11 queries – RQ1 through RQ11 – that are compliant with the query templates used in [23]. 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, REGAL did not complete a few queries, denoted by DNC in the figure.

Due to these large qualitative and quantitative differences wrt QRE, we present only the UNMASQUE results in the remaining experiments of this section.

Comparison with TALOS (Experiments on UCI datasets) We also conducted experiments on two small UCI [51] datasets – specifically Adult and Baseball. Adult is a single-relation data set that has been used in many classification works, while Baseball is a more complex, multi-relation database containing player performance statistics over 135 years.

We ran UNMASQUE on those queries where TALOS outputs the exact query as used in the respective input. The queries are listed in [25]. For all the queries, UNMASQUE extraction was significantly faster as highlighted in Figure 14. The numbers for TALOS were taken from [25].

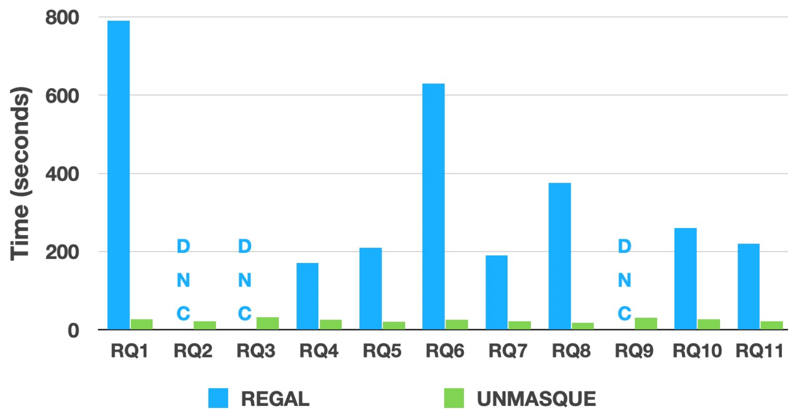


Figure 13: Comparison with QRE (5 GB)

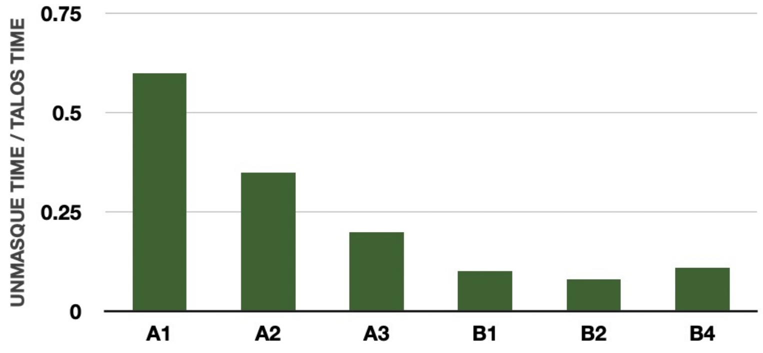


Figure 14: Comparison with TALOS

6.2 Hidden SQL Queries

Our primary extraction experiments were conducted on a basal suite of EQC^{-H}-compliant but complex SPJGAOL queries, derived from the following popular benchmarks: (a) **TPC-H** (12 queries), (b) **TPC-DS** (7 queries), and (c) **JOB** [37] (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 first present experiment results on TPC-H benchmark with detailed explanation and then summarize results for TPC-DS and JOB benchmarks.

6.2.1 Hidden SQL Queries - TPC-H (Without Sampling)

For these experiments, we used a basal suite of SPJGAOL warehouse queries, which are derived from the TPC-H benchmark and compliant with EQC^{-H}. They are similar in complexity to the **Q3** running example, and are listed in the Appendix. For convenience, we hereafter refer to them as Qx, where x is their associated TPC-H query identifier. Each query was passed through a Cpp program that embedded the query in a separate executable. These executables formed the input to UNMASQUE, which has been implemented in Python, and were invoked on the TPC-H database, assuring a populated result. UNMASQUE's ability to non-invasively extract these queries was assessed on a 100 GB version of the TPC-H benchmark, and to profile its scaling capacity, also on a 1 TB environment.

Correctness We compared the $\mathcal{Q}_E$ output by UNMASQUE on the above $\mathcal{Q}_H$ suite with the original queries. Specifically, we verified, both manually and empirically with the automated Checker compo-

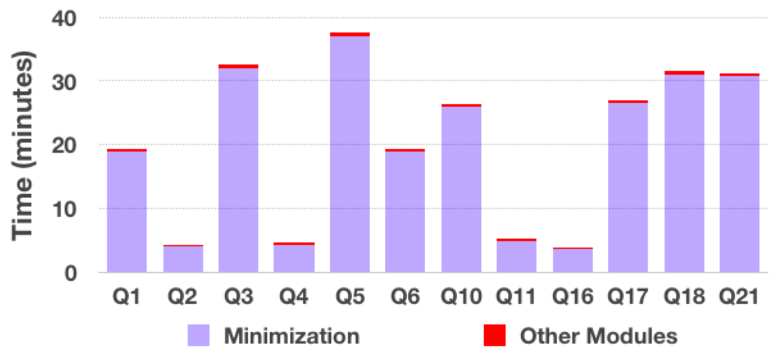


Figure 15: Hidden Query Extraction Time (100 GB) – no sampling

nent of the pipeline, that the extracted queries were semantically identical to their hidden sources.

The Checker implements two kinds of checks: First, a number of randomized large databases are created on which both the executable and the extracted query are run. The results are compared through set difference, and a non-zero outcome indicates an error. Physical ordering is checked by computing a position-dependent checksum function on the entire result.

Second, we leverage the popular XData tool [12] which is intended for checking the correctness of student queries wrt to the instructor's solution by constructing databases that are able to detect even subtle differences. In our context, the application is mapped to the instructor's solution, and the extracted query to the student's attempt, and it is then verified that they are identical.

Efficiency The total end-to-end time taken to extract each of the twelve queries on the 100 GB TPC-H database instance is shown in the bar-chart of Figure 15. In addition, the breakup of the primary pipeline contributors to the total time is also shown in the figure.

We first observe that the extraction times are practical for offline analysis environments, with all extractions being completed within 40 minutes. Secondly, there is a wide variation in the extraction times, ranging from 4 minutes (e.g. Q2) to almost 40 minutes (e.g. Q5). 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 (blue color), take up the lion's share of the extraction time, the remaining modules (red color) collectively completing within a few seconds. For instance, for Q5 which consumed around 37.2 minutes overall, the MINIMIZER expended around 37 minutes, and only a paltry 12 seconds was taken by all other modules combined.

The extreme skew is because the MINIMIZER module operates on the original large database, whereas, as described in Sections 4 and 5, the remaining modules work on miniscule mutations or synthetic constructions that contain just a handful of rows. Interestingly, although the executable $\mathcal{E}$ was invoked a *few hundred* times during the operation of these modules, the execution times in these invocations was negligible due to the tiny database sizes.

6.2.2 Hidden SQL Queries - TPC-H (With Sampling)

We now show how even the minimization step's efficiency could be substantially improved. Specifically, instead of executing MINIMIZER on the *entire* original database, *sampling* methods that are natively available in most database systems could be leveraged as a pre-processor to quickly reduce

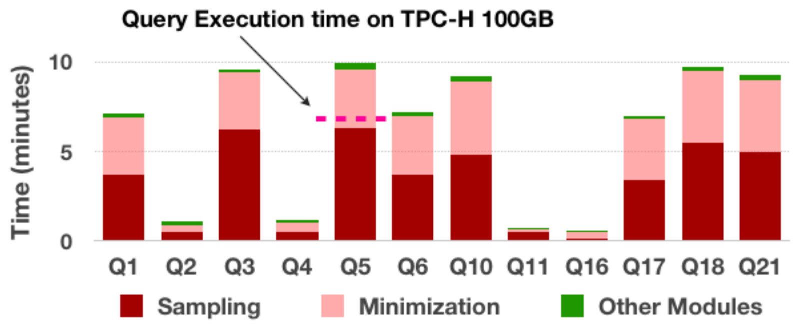


Figure 16: Optimized Hidden Query Extraction Time (TPC-H 100GB)

the initial size. In our implementation, we iteratively sample the large-sized tables, one-by-one in decreasing size order, until a populated result is obtained.

The sampling is done using the following SQL construct:

select * **from** table **where** random() $<$ 0.SZ ;

which creates a random sample that is SZ percent relative to the original table size. An interesting optimization problem arises here – if SZ is set too low, the sampling may require several failed iterations before producing a populated result. On the other hand, if SZ is set too large, unnecessary overheads are incurred even if the sampling is successful on the first attempt. Currently, we have found a heuristic setting of **SZ = 2%** in terms of number of rows to consistently achieve both fast convergence (within two iterations) and low overheads. In our future work, we intend to theoretically investigate the tuning of the sample size parameter.

The revised total execution times after incorporating this optimization, are shown in Figure 16, along with the module-wise breakups. We see here that all the queries are now successfully identified in *less than 10 minutes*, substantially lower as compared to Figure 15. Further, the FROM clause takes virtually no time, as expected, and is therefore included in the Other Modules category (green color). And in the MINIMIZER, the preprocessing effort spent on sampling (maroon color) takes the majority of the time, but greatly speeds up the subsequent recursive partitioning (pink color).

An alternative testimonial to UNMASQUE's efficiency is obtained when we compare 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, a single execution of $Q5$ on the 100GB database took around 6.5 minutes, shown by the red dashed line in Figure 16, while the extraction time was just under 10 minutes.

Finally, as an aside, it may be surmised that popular database subsetting tools, such as Jailer [35] or Condenser [49], could be invoked instead of the above sampling-based approach to constructively achieve a populated result. However, this is not really the case due to the following reasons: Firstly, these tools do not scale well to large databases – for instance, Jailer did not even complete on our 100 GB TPC-H database! Secondly, although they guarantee referential integrity, they cannot guarantee that the subset will adhere to the *filter predicates* – due to the hidden nature of the query. So, even with these tools, a trial-and-error approach would have to be implemented to obtain a populated result.

Scaling Profile To explicitly assess the ability of UNMASQUE to scale to larger databases, we also conducted the same set of extraction experiments on a **1 TB** instance of the TPC-H database. The results of these experiments, which included all optimizations, are shown in Figure 17. We see here that all extractions were completed in less than 25 minutes each.

We also conducted the same set of extraction experiments on a suite of different-sized instances of the TPC-H database, starting from 200 GB and going up to 1 TB in increments of 200 GB. A sample

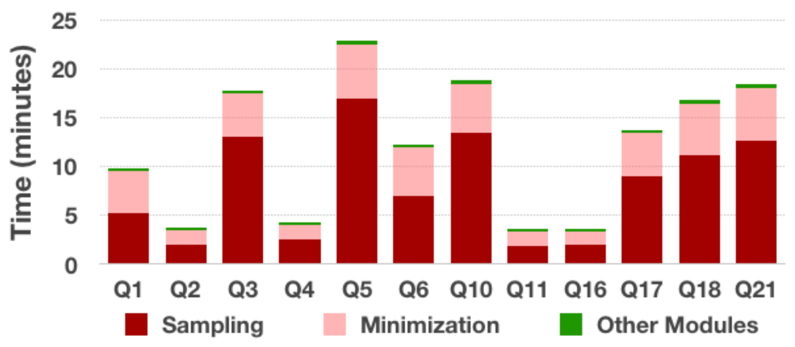


Figure 17: Optimized Hidden Query Extraction Time (TPC-H 1 TB)

result corresponding to Q5, the most difficult extraction, is shown in Figure 18. 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.

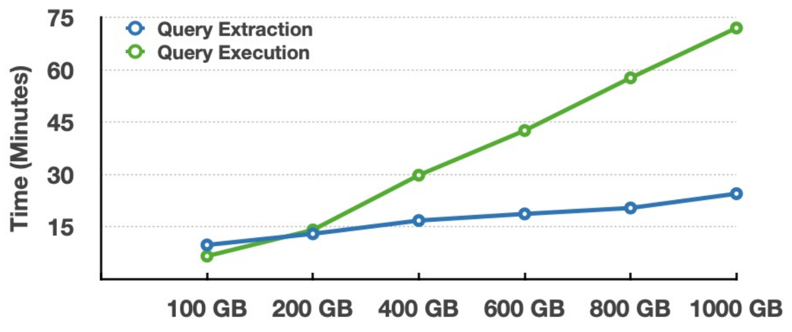


Figure 18: Extraction Scaling Profile Q5 (TPC-H)

6.2.3 TPC-DS Results

We have also run UNMASQUE on queries based on the TPC-DS benchmark, with a 100 GB version of the database hosted on PostgreSQL. The queries are listed in the Appendix.

The bar-chart in Figure 20 shows the time taken to extract 7 such queries (along with their TPC-DS identifier numbers). We can see that all the queries were extracted within 4 minutes. It may appear surprising at first that the time taken in this case is lesser than the time for TPC-H queries and also, that the variation among queries is small. The reason is that the table sizes in TPC-DS are not as skewed as in TPC-H– in particular, no table in TPC-DS is as huge as the lineitem table of TPC-H.

6.2.4 Join Order Benchmark (JOB) Results

We have also run UNMASQUE on queries based on Join Order Benchmark [37], constructed on the real-world IMDB movie dataset [36]. The 11 queries evaluated here, picked from the hundred-plus queries in the benchmark, 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 the complexity, they were all correctly extracted in less than **3 minutes** apiece, as shown in Figure 19. Again the reduction of the initial database size via sampling and partitioning takes the lion's share of the extraction, with the remaining modules collectively completing in less than a minute.

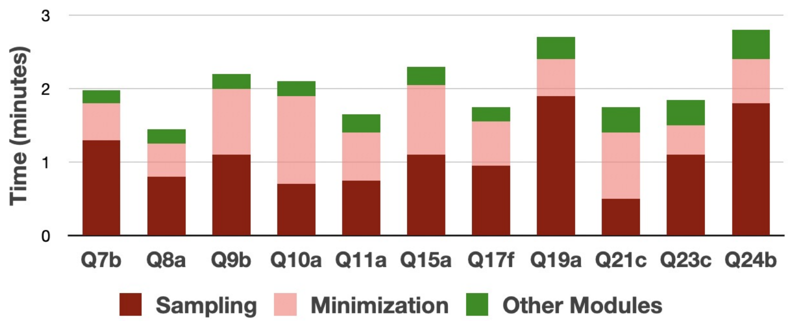


Figure 19: Hidden Query Extraction Time (JOB)

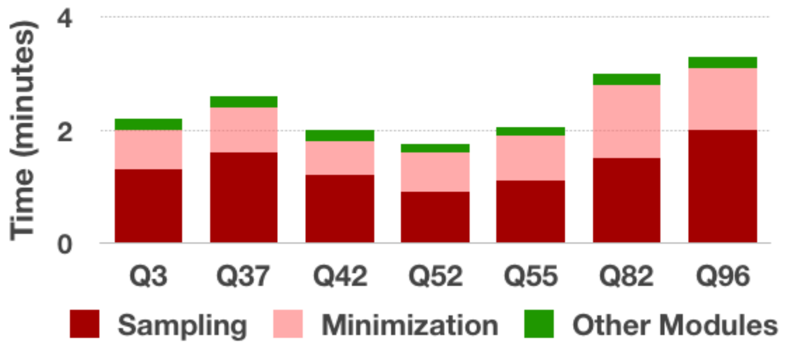


Figure 20: Hidden Query Extraction Time (TPC-DS 100 GB)

6.3 Hidden Imperative Code

Our second set of experiments evaluated applications hosting imperative code. Here we considered a) The popular **Enki** [38] and **Blog** [45] blogging applications, both built with Ruby on Rails, each of which has a variety of commands that enable bloggers to navigate pages, posts and comments, (b) **Wilos** [52], a process orchestration software based on the Hibernate ORM and (c) **RUBiS** (auction site benchmark) [39], a popular data framework used in recent literature (e.g. *Aggify* [9]).

Enki and Blog The Enki and Blog servers receive HTTP requests, interact with the database accordingly, and respond the client with an HTML page that contains the data retrieved. Enki uses a total of eight database tables and Blog uses two database tables. We created a synthetic database of 10 MB size which gives non-empty result for each of these commands. Along with UNMASQUE, we used Selenium [41] to send an HTTP request and receive the results in HTML page from which the database results are automatically extracted.

Since native data is not publicly available, we created a synthetic 10MB database that provided populated results for all these commands. We found that for Enki, 14 out of 17 and for Blog 2 out of 2 commands were extracted (except insert, update, etc.). Table 3 shows the SQL queries extracted w.r.t. the commands (five commands have been omitted as those were simple table scans). The queries corresponding to remaining three commands did not belong to EQC and only SPJ part was extracted correctly for them. We manually verified that all the commands in Table 3 were extracted correctly. As a sample instance, consider the *"get latest posts by tag"* command, a sample few line snippet of which is outlined in Figure 21a. UNMASQUE output corresponding to the function *"find_recent"* is shown in Figure 21b, and was produced in just 2.5 seconds.

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

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

```
Select min(posts.title), min(posts.body), max(posts.published_at),
count(*), 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 taggings.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 Query (cur_timestamp is a constant)**

Figure 21: Imperative to SQL Translation

Command	Application	Extracted SQL Complexity	Time
get admin comments	Enki	Project, Join, OrderBy, Limit	1.2 sec
get admin pages	Enki	Project, OrderBy, Limit	1 sec
get admin pages id	Enki	Filter, Project, Limit	1 sec
get admin posts	Enki	Project, Join, GroupBy, OrderBy, Limit	2.5 sec
get admin posts id	Enki	Filter, Project, Limit	1 sec
get admin comments id	Enki	Filter, Project, Limit	1 sec
get admin undo items	Enki	Filter, Order by, Limit	.5 sec
get latest posts	Enki	Filter, Project, Join, Filter, GroupBy, Order By, Limit	1.5 sec
get user posts	Enki	Filter, Project, Join, Filter, Group By, Order By, Limit	2.5 sec
get latest posts by tag	Enki	Filter, Project, Join, Filter, GroupBy, OrderBy, Limit	2.5 sec
get article for id	Blog	Select, Project, join	1 sec

Table 3: Imperative to SQL Translation - Enki & Blog (10 MB)

File (Function Line No.)	Extracted SQL Complexity	Time (sec)
ActivityService (401)	Project, Join	1.2
ActivityService (328)	Project, Join, Group By	1.5
Guidance Service (140)	Project, Join, Group By	1.3
Guidance Service (154)	Project, Join, Group By	2.0
ProjectService (266)	Project, Join, Order By	1.8
ProjectService (297)	Filter, Project, Join, Group By	2.4
ProjectService (394)	Filter, Project, Join, Group By	2.5
ProjectService (410)	Filter, Project, Join, Group By	2.3
ProjectService* (248)	Filter, Project	.5
AffectedtoDao* (13)	Filter, Project	.4
ConcreteActivityService (133)	Project, Join , Group By	2.3
ConcreteRoleAffectationService (55)	Project, Join	2.5
ConcreteRoleDescriptorService (181)	Project, Join, Group By	2.1
ConcreteWorkProductDescriptorService(236)	Project, Join, Group By	1.9
IterationService (103)	Project, Join, Group By	2.0
LoginService* (103)	Filter, Project, toUpper() function	1.2
LoginService* (83)	Filter, Project, ignoreCase() function	1.2
ParticipantService (119)	Project, Join, Group By	1.6
ParticipantService (266)	Project, Filter, Join, Group By	1.6
PhaseService (98)	Project, Join, Group By	1.3
RoleDao (15)	Project, Filter, Aggregation	1.3
WorkProductsExpTableBean* (974)	Project, Join, Aggregation, Having	6.4

Table 4: Imperative to SQL Translation - Wilos (10 MB)

Function	Extracted SQL Complexity	Time(sec)	Function	Extracted SQL Complexity	Time(sec)
CommentsToUser	Filter, Project	0.4	CommentsByUser	Filter, Project	0.4
SoldItems	Filter, Project	0.45	OnSaleItems	Filter, Project	0.45
BoughtItems	Filter, Project	0.45	WonItems	Filter, Project, Join, Group By	.9
UserBids	Filter, Project, Join, Group By	1.4	BidByItem	Filter, Project, Order By	0.55
NumBidByItem	Filter, Project, Aggregation	0.5	MaxItemBid	Filter, Project, Aggregation	0.5
ItemById	Filter, Project	0.4	UserById	Filter, Project	0.35
UserByNickname	Filter, Project	0.35	NicknameExists	Filter, Project, Aggregation	0.5
RegionIdForName	Filter, Project	0.35	ItemsByRegion	Filter, Project, Join	1.1
ItemsByCatagory	Filter, Project, Join	1.1			

Table 5: Imperative to SQL Translation - RUBiS (10 MB)

Wilos This Java-based application has been previously used to showcase the potential of imperative-to-SQL conversion tools such as DBridge [3] and QBS [4]. There are 33 code snippets listed in [4], 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[4]since Wilos uses only this specific row as input from the source table.

We verified that UNMASQUE was able to correctly extract the equivalent queries for all 22 functions, accomplishing these tasks within a few seconds. Table 4 shows detailed results for the functions with extracted query complexities. There were some functions returning TRUE/FALSE and UNMASQUE was modified to treat TRUE as non-empty output and FALSE as empty output. The last entry in Table 4 is an interesting case where the query extracted by basic pipeline was rejected by the checker module and then the function was passed to having clause pipeline (explained later in Section 7).

RUBiS Built on top of Java, PHP7 and Mysql database server, RUBiS is a bidding system modeled after *ebay.com*. Similar to Wilos, 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[4]. There are 17 code snippets (functions) which interact with database and UNMASQUE was able to extract the equivalent SQL for all the 17 functions. Table 5 shows detailed results for the functions with extracted query complexities.

Performance Comparisons The efficiency benefits of imperative-to-SQL transformations are well established in the literature [3, 4] and have led to their incorporation in mainstream database products (e.g. Froid in SQL Server). We have observed the same with the Enki commands – this is quantified in Table 6, where for each of the commands, we show its extraction complexity and the *speedup* of the extracted SQL query relative to the original imperative code on a 100 GB instance (with indices on key columns). We see here that the performance gains are often significant – in fact, for the sample Enki command "get latest posts by tag" shown in Figure 21, the speedup is by more than an *order-of-magnitude*.

7 Having Clause Extraction

Thus far, we had deliberately set aside discussion of the HAVING clause. The reason is that this clause is especially difficult to extract, stemming from its close similarity to filter predicates in the WHERE clause – this difficulty has led to it not being considered in the prior QRE literature as well. The good

[4]The value for key columns were taken as 1 in the input row to make it consistent with Generation Pipeline algorithms.

Command	Extracted SQL Complexity	Speedup (SQL/Imperative)
get admin comments	Project, Join, OrderBy, Limit	5.0
get admin pages	Project, OrderBy, Limit	1.5
get admin pages id	Filter, Project, Limit	1.8
get admin posts	Project, Join, GroupBy, Aggregation, OrderBy, Limit	7.7
get admin comments id	Filter, Project, Limit	1.25
get admin posts id	Filter, Project, Limit	1.4
get admin undo items	Project, Order by, Limit	1.5
get latest posts	Filter, Project, Join, GroupBy, Aggregation, Order By, Limit	6.7
get user posts	Filter, Project, Join, Group By, Aggregation, Order By, Limit	10.0
get latest posts by tag	Filter, Project, Join, GroupBy, Aggregation, OrderBy, Limit	12.5

Table 6: Imperative to SQL Performance gain - Enki (100 GB)

news is that we have been able to devise an extraction technique under a few assumptions, the primary one being that the attribute sets in F_E and H_E are disjoint[5] However, incorporating this approach entails a significant reworking of the UNMASQUE pipeline, as well as modified algorithms for some of the modules. Specifically, the extraction of filter predicates is now delayed to *after* the GROUPBY module, and the implementations of the FILTERPREDICATE and GROUPBY modules are altered.

In addition to the assumptions in Section 3, the SPJGA queries with having clause should satisfy the following conditions.

1. As mentioned above, the attributes involved in filter predicate in the Having clause and in the Where clause are disjoint.
2. Each attribute has at most one aggregation in the Having clause predicates.
3. The values in the Having clause predicates do not exceed the bounds of their corresponding data types.

Our discussion here is limited to the HAVING clause on integer attributes. However, the queries with textual attributes (and LIKE operator) can also be handled, in a manner similar to those described in the previous sections.

7.1 Initial Modules

The From Clause detection, Database Sampling and Join Graph Detection are performed in the same manner as described in Sections 4.1, 4.2.3 and 4.3, respectively.

7.2 Database Minimization

The database minimization step has to be modified to cater to the H_E clause. Specifically, we leverage the following lemma:
Lemma 8: *For EQC, there always exists a D_{min} such that the output of the SPJ core of the query constitutes a **single group** (as per grouping attributes of the query), and the final output contains only a **single row**.*

Proof. We prove the above claim by contradiction. Let us say that there exists a query Q in EQC such that the final result for a database instance D contains $m : m > 1$ rows and removing any row from any table in the query gives an empty result (i.e. D is a minimal database for Q). According to the query

[5]This assumption holds for all the queries of the TPC-DS benchmark.

structure of EQC, if $H_E \neq \phi$, it implies that $G_E \neq \phi$. Now as the final result contains m rows, the intermediate SPJ result must have $\geq m$ rows with two different combinations of values in the columns of G_E. Let us divide the rows in intermediate result based on the combination of values in the columns of G_E. If we keep only those database rows which correspond to just one set of these rows, given the structure of the query in EQC, we should get a populated output having a single row. As we are not able to remove even a single row from any table of database D which appears in the From clause of query Q, it implies that Q does not follow the structure defined for the query in EQC and hence $Q \notin$ EQC. A contradiction. □

7.2.1 Reducing $\mathcal{D}_I$ to D_{min}

The minimization is done in the following manner: Let t be a table in T_E. Initially, for each attribute in t, the frequency of each value is calculated. Let $f_{A,j}$ denotes the maximum value of frequency with value j in attribute $t.A$. After that, the rows corresponding to $f_{A,j}$ are preserved, removing all other rows. If a non-empty output is produced, the preserved rows form the new table-content on which frequency values are recalculated and the same procedure is repeated. If an empty output is produced, the same procedure is applied with the value having the next maximum frequency. This procedure is repeated until no further reductions are possible in t. Once t is reduced, all the tables connected to it in the join graph are reduced to contain only those rows which satisfy the join condition.

The above procedure is repeatedly applied to each table in T_E until the database cannot be reduced any further. The idea behind the step of preserving a particular value of the attribute is as following: if $t.A$ is a grouping attribute, it will contain a single value in the reduced database instance. Further, our heuristic of first selecting the value with the maximum frequency is because it ensures that a relatively large number of rows are preserved in each iteration, thereby increasing the possibility of a populated output.

An important point to note here is that the number of rows in the minimized database for a query in EQC is a function of both the original database contents and the specific Having clause predicate. For example, with a Having clause predicate *count(*)* $\geq$ *m*, the maximum number of rows in each table of the minimized database is m.

Also note that if the query belongs to EQC^{-H}, the final database will definitely be a one-row database. However, we may get a one-row database even if the query belongs to EQC. For now, we assume that the reduced database is a multi-row database, and defer the special case of one-row database to Section 7.8.

7.3 Group By Attributes

It is clear by D_{min} construction that any column with two or more unique s-values cannot be a grouping attribute as it would have created two different groups in the output. So, in order to identify the attributes in G_E, we first identify the columns in D_{min} that have a single value in all the rows. For each such column $t.A$ with value v_1, we insert a duplicate row in table t with only difference being that $t.A = v_2$, where $v_2 \neq v_1$. To ensure that v_2 is an s-value, we do this twice, first with $v_2 = v_1 + 1$ and then with $v_2 = v_1 - 1$ and run $\mathcal{E}$ each time. An output with two rows in either of these cases indicates $t.A$ to be an element of G_E.

7.4 Filter and Having Predicate Extraction

First we identify possible filter on each G_E column using a similar technique to that of Section 4.4, the only change being that mutation is applied to all rows of the table.

Next, the filters on non-grouping attributes are identified. For this purpose, we utilize the observation that a filter constraint $a \leq A \leq b$ can be equivalently re-written as HAVING clause predicates: $a \leq min(A)$ and $max(A) \leq b$. This means that both F_E and H_E predicates can be extracted in a *unified* manner, followed by a separation into the two categories, as explained below. Further, each H_E predicate can be generically represented as $a \leq agg(A) \leq b$, separable into two components – $agg(A) \geq a$ (Lower Bound) and $agg(A) \leq b$ (Upper Bound) – that can be identified individually. Before extracting Lower and Upper Bounds, we do some preprocessing as mentioned below.

7.4.1 Preprocessing

To detect the having clause condition on a database column $t.A$, we change its values in the table t, such that only one row of the output group is affected. However, if a foreign key column of t maps to a primary key column of another table t' in the join graph and values in the foreign key column are not unique, one change in the table will affect multiple places in the output group. So we transform the tables in a way such that all key values in all the tables are unique and there is one-to-one relationship between the key columns across tables. This can be done by traversing the join graph and duplicating rows in the table having primary key column (here t') with new key values and updating these key values in the corresponding rows of table t. For example, let $T_1[(1, `a', 2), (2, `b', 2)]$ be a table with two rows and $T_2[(2, `c')]$ be a table with a single row where the last attribute of $T_1$ refers to the first attribute of $T_2$. Here, $T_1$ can be seen as table $t$ and $T_2$ can be seen as table $t'$ as per the above description. Then, these tables are transformed as $T_1[(1, `a', 1), (2, `b', 2)]$ and $T_2[(1, `c'), (2, `c')]$. Note that both the joins (before and after transformation) produce same output except the key column contents.

7.4.2 Identifying the Bounds

Let $[i_1, i_2]$ be the integer range, and let $(a_1, a_2, ..., a_n)$ be the values in attribute A in non-decreasing order. Wlog, let us assume a_i is the value in attribute A in the i^{th} row.

Lower Bound We first define the term r_l and v_l. Starting from 1 to n, if we keep decreasing the value of a_i to i_1, r_l denotes the first row, in which the values in A can not be decreased to i_1 without losing the output. Also, $r_l = none$ if values in all the rows can be decreased to i_1. Further, if $r_l \neq none$, v_l denotes the minimum value in row r_l which can be present without losing the output. The following algorithm is used to get r_l and v_l.

Now, the following two cases arise:

Case 1: $r_l = none$**.** In this case, there is no lower bound condition on A. The reason is that, we were able to reduce value in every row to minimum possible value without loosing the output.

Case 2: $r_l \neq none$**.** If $r_l \neq 1$ and $r_l \neq n$, there is a having clause predicate on A with either $sum() \geq val_1$ or $avg() \geq val_1$. The reason is that, if there were a condition $min(A) \geq v_l'$, the value of r_l should have been 1. Similarly, if here were a condition $max(A) \geq v_l'$, the value of r_l should have been n. Now, if $r_l = 1$, the aggregations in the filter predicate may be $sum()$, $avg()$ or $min()$. To differentiate amongst these, we decrease the value in the first row by 1 and increase the value in any

Algorithm 3: Getting r_l and v_l for lower bound

```
$r_l = none, v_l = none$
for $i$ in range 1 to $n$ do
    $v_l \leftarrow$ the minimum value in $[i_1, a_i]$ which gives non-empty result.
    if $v_l = i_1$ then
        Replace $a_i$ with $i_1$ in the database
        $v_l = none$ continue
    end
    Replace $a_i$ with $v_l$ in the database
    $r_l = i$
    break
end
```

other row by 1. This makes sure that the $sum(A)$ and $avg(A)$ does not change while changing $min(A)$. If we get an output, the filter is either $sum() \geq v'_l$ or $avg() \geq v'_l$ otherwise it is $min(A) \geq v'_l$. A similar method can be used to differentiate amongst $sum()$, $avg()$ or $max()$ when $r_l = n$. In case of $min()$ or $max()$ aggregation, the corresponding filter value v'_l will be the v_l obtained from the algorithm. However, in case of $sum()$ or $avg()$ aggregation, v'_l can be calculated from the table content at the end of algorithm 3.

Upper Bound To find the upper bound on A, a similar approach can be used. Here, we define terms r_u and v_u as follows. Starting from n to 1, if we keep increasing the value of a_i to i_2, r_u denotes the first row, in which the values in A can not be increased to i_2 without losing the output. Also, $r_u = none$ if values in all the rows can be increased to i_2. Further, if $r_u \neq none$, v_u denotes the maximum value in row r_u which can be present without losing the output. The following algorithm is applied to get the r_u and v_u.

Algorithm 4: Getting r_u and v_u for right filter

```
$r_u = none, v_u = none$
for $i$ in range n to 1 do
    $v_u \leftarrow$ the maximum value in $[a_i, i_2]$ which gives non-empty result.
    if $v_u = i_2$ then
        Replace $a_i$ with $i_2$ in the database
        $v_u = none$
        continue
    end
    Replace $a_i$ with $v_u$ in the database
    $r_u = i$
    break
end
```

After getting r_u and v_u, upper bound can be found in a similar way using the following two cases:

Case 1: $r_u = none$**.** In this case, there is no upper bound condition on A.

Case 2: $r_u \neq none$**.** If $r_u \neq 1$ and $r_u \neq n$, there is a having clause predicate on A with either $sum() \leq v'_u$ or $avg() \leq v'_u$. Now, if $r_u = n$, the aggregations in the filter predicate may be $sum()$,

$avg()$ or $max()$. To differentiate amongst these, we increase the value in the n^{th} row by 1 and decrease the value in any other row by 1. This makes sure that the $sum(A)$ and $avg(A)$ doesn't change while changing $max(A)$. If we get an output, the filter is either $sum() \leq v'_u$ or $avg() \leq v'_u$ otherwise it is $max(A) \leq v'_u$. A similar method can be used to differentiate amongst $sum()$, $avg()$ or $min()$ when $r_u = 1$. In case of $min()$ or $max()$ aggregation, the corresponding filter value v'_u will be the v_u obtained from the algorithm. However, in case of $sum()$ or $avg()$ aggregation, v'_u can be calculated from the table content at the end of algorithm 3.

Note that we have not yet differentiated between the filters on $sum()$ and filters on $avg()$. Let the current average of the values in column A be a. To differentiate between the two for an attribute A, we insert a row in the table such that the column A is assigned value 0 (if operator is $\geq$) or it is assigned value a (if operator is $\leq$). All the group by attributes are assigned a single value in all the rows, the other attributes with $sum()$ or $avg()$ filter are assigned $null$ in the new row and all other attributes get any value satisfying the filter predicate. This construction ensures that the output state is directly dependent on the changes made in attribute A. Based on the output on this new database, we can differentiate between $sum(A)$ and $avg(A)$. Further if the average is a floating point number, we can refine it using binary search assuming fixed precision.

7.4.3 Predicate Separation

Finally, all the H_E predicates of the form $min(A) \geq a$ and $max(A) \leq a$ are converted into corresponding filter predicates. This is from the perspective of efficient query execution, since it is usually recommended that filters should be applied as early as possible.

Finally, after identifying all other filters, a Having predicate with *count()* can be processed in a manner analogous to finding LIMIT in Section 5.4.

7.5 Projection Clause

In the absence of a Having clause filter of type $val_1 \leq sum() \leq val_2$, the techniques defined in Section 4.5 can be used to detect scalar functions in the Having clause by placing a single unique value in every row of each column and normalizing the final coefficients by the number of rows produced after the joins and filters. However, in the presence of such filters, we may not be able to do so as we may not have much choice for arbitrary unique values in the column. In such a case, we may get an under-determined system of equations and any solution can be treated as the function.

7.6 Generation Pipeline (Except G_E)

If there is no filter of the type $count(*) \geq m$ in the Having clause, we can create a one-row database satisfying all the filters. Hence, procedures similar to those for EQC^{-H} can be used here as well. **In such a case, the sizes of the synthetic databases will be same as in the Generation Pipeline of Section 5**.

In case a filter of the above type is present, an additional constraint of number of rows is added while generating databases while the underlying principle remains the same. **In this case, the sizes of the synthetic databases will be a linear function of** m.

7.7 Performance

Although the extraction process is materially different, our experiments show that queries featuring the HAVING clause are also handled efficiently. For instance, when the predicate

HAVING min $(l\_extendedprice) > 1000$

was added to **Q3**, extraction on the 100 GB instance was completed in 10 min, only marginally more than the 9.3 min taken for the original query. Also, the number of rows in the minimized database were less than 100 for each table. A similar behavior was seen for all our other queries as well.

7.8 Special Case of One-row database for SPJGHA[OL] query

After database minimization, we may get a one-row database for a SPJGA query with Having clause. However, to detect this clause, we need a database such that the intermediate output of SPJ part contains at least two rows. In such a case, we first detect the group by clause as mentioned in Section 7.3. After that, in each table, we insert the existing row again with a different key value. If we get a two-row output, we conclude the query is an SPJ query. If we get a single row output, we now have a single group database with more than one row in the intermediate SPJ output.

However, we may get an empty output as well. For instance, consider an attribute A containing a value 6 in the database, and the query with a Having clause predicate defined as $sum(A) < 10$. In such a case, replicating the value will make $sum(A) = 12$ and hence we will not get any output. As there is no way of knowing beforehand which attribute caused the output to be non-empty, we iteratively place $null$ values in a progressively larger subset of attributes, starting from individual elements, until a populated output is obtained.

8 Query Coverage Extensions

As promised in Section 3, we now discuss how the UNMASQUE extraction process can be extended to databases having non-integral key columns. After that, we speculate on future prospects for extending our current techniques to more complex query structures.

8.1 Non-integral Key Attributes

In Section 3, we assumed that all keys are positive integer values. We now discuss how to handle keys from other domains since many applications (e.g. Wilos [52]) use non-integral keys as identifiers in the database tables. In this design, we make the mild assumption that the domain of each key attribute contains at least two different values.

To handle the above framework, the following changes are required:

1. In Mutation Pipeline, only the join predicate extraction module require changes. In this module, instead of negating the values of the columns in C_1 (refer Section 4.3), we choose two different fixed values (say p and q) from the domain of the key attribute and assign p to the columns in C_1 and q to the columns in C_2.

2. For every module in Generation Pipeline, we again take two different fixed values (say p and q) from the domain of the key attribute. Then, all the assignments that use fixed value 1 are replaced with value p and all the the assignments that use fixed value 2 are replaced with value q.

8.2 Discussion

A natural question to ask at this point is whether it appears feasible to extend the scope of our extraction process to a broader range of common SQL constructs – for instance, outer-joins, disjunctions and nested queries. As mentioned previously, none of these constructs are handled by the current set of QRE tools. However, based on some preliminary investigation, it appears that outer-joins and disjunctions could eventually be extracted under some restrictions – for instance, the IN operator can be handled if it is known that the database includes all constants that appear in the clause. Nested queries, however, pose a formidable challenge that perhaps requires novel technology. In this context, an interesting possibility is the potential use of machine-learning techniques for complex extractions.

9 Related Work

As mentioned in the Introduction, HQE is a variant of QRE, a problem that has seen considerable activity in recent years [29, 25, 23, 22, 11, 4, 1, 16]. A critical difference, however, is that the QRE goal is to find, given an *instance* of a $\{\mathcal{D},\mathcal{R}\}$ pair, *some* satisfying query $\mathcal{Q}$ such that $\mathcal{Q}(\mathcal{D}) = \mathcal{R}$. In contrast, the HQE objective is to find the *specific* $\mathcal{Q}_H$ embedded in application $\mathcal{A}$, which therefore produces the correct result on *all* databases, not just on the instance $\mathcal{D}$. In other words, there is a ground truth against which the extraction is to be evaluated.

Although using database generation as a core mechanism, UNMASQUE is different from approaches like XData [19] which generate databases for *distinguishing* mutations of a known correct query. In contrast, our goal is to create database mutations that help to *construct* the correct query. In short, the former is intended for testing/grading purposes, whereas we wish to reconstruct applications in enterprise settings. However, we do use XData to verify extraction correctness in the concluding Checker module of our pipeline, as explained in Section 6.

UNMASQUE is also related to but distinct from automated imperative-to-SQL conversion tools such as DBridge [3, 7] and its commercial version, Froid [18]. These tools are host-language-specific (e.g. T-SQL in the case of Froid), and require support for the special APPLY or LATERAL operators which are not 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 application and platform-independent.

Our objectives are different to Query-by-Output [15] which aims to interactively help naive users construct the desired SQL queries. Their setup assumes that users are able to observe results on candidate queries and determine whether or not they contain the desired answer, an approach that is not easily scalable to large or aggregated result sets. Moreover, similar to QRE, there is no ground truth for comparison.

Finally, related issues have also received considerable attention in the programming languages community (e.g. [4, 20, 30, 27]). The work done in [4] provides conversion from imperative logic to SQL for limited scenarios using query synthesis rather than program analysis. The approaches in [30, 27] are similar to QRE where a query is synthesized based on input-output examples. Lastly, [20] attempts to find equivalent functions from program executables predicated on intimate knowledge of the application semantics and its database interactions.

Detailed Comparison With QRE As mentioned in the Introduction, HQE is a conceptual variant of the long-standing QRE problem. However, our UNMASQUE solution has been constructed from first principles. The reasons for doing so are explained here, using REGAL as an exemplar for comparison, from both pipeline and component perspectives.

The extraction pipeline sequence in REGAL identifies candidates for joins and projections prior to the other clauses. Whereas, UNMASQUE has to first identify the filters in advance so that the synthetic databases in the generation pipeline are populated with s-values– that is, values compatible with the where clause predicates. Secondly, these generated databases have to be carefully constructed to produce *known*, albeit invisible, intermediate behavior. Thirdly, since UNMASQUE requires hundreds of application invocations of $\mathcal{Q}_H$ to set up its input-output examples, it perforce needs to work on tiny databases that will allow these numerous executions to complete in reasonable time. Finally, the operator scope of REGAL is not only considerably limited in comparison to UNMASQUE, but these limitations are fundamental due to the restricted availability of input-output example.

After that, a materialized view corresponding to the join-projection clauses is created for 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 column will lead to selection of wrong GROUP BY-AGGREGATION candidate as seen in Figure 2b.

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 2b. 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, as quantified in Section 6, REGAL requires considerable memory due to its heavy-weight internal data structures, and from a time perspective, only small input databases can be processed due to the large exploration space.

10 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, and covering both explicit and implicit application opacity. Addressing the HQE problem proved to be challenging due to the inherent complexities and acute dependencies between the various query clauses. As a first-cut solution, we presented the UNMASQUE algorithm, which non-invasively and efficiently extracts a basal class of hidden SPJGHAOL queries through a pipeline that leverages database mutation and generation techniques to identify the clauses in a systematic manner.

Potent optimizations related to database minimization and order detection were incorporated to reduce the overheads of the extraction process. Specifically, for the most part, the extraction pipeline works on minuscule databases designed to contain only a handful of rows. The effects of these optimizations were visible in our experimental results which demonstrated that query extraction could be completed in times comparable with normal query response times in spite of a large number of executable invocations.

In our future work, we intend to conduct a mathematical analysis to help choose the appropriate SZ setting for sampling. We also wish to investigate extending the scope of query coverage to more constructs, especially disjunctions and outerjoins. Finally, and more fundamentally, characterizing the extractive power of non-invasive techniques is an open theoretical problem.

References

[1] A. Bonifati, R. Ciucanu and S. Staworko. Learning Join Queries from User Examples. *ACM TODS*, 40(4), 2016.

[2] B. Chandra. Automated Testing and Grading of SQL Queries. *PhD Thesis, CSE, IIT Bombay*, 2019.

[3] M. Chavan, R. Guravannavar, K. Ramachandra and S. Sudarshan. DBridge: A program Rewrite Tool for Set-Oriented Query Execution. In *Proc. of IEEE ICDE Conf.*, 2011.

[4] A. Cheung, A. Solar-Lezama and S. Madden. Optimizing Database-Backed Applications with Query Synthesis. In *Proc. of ACM PLDI Conf.*, 2013.

[5] P. da Silva. SQUARES : A SQL Synthesizer Using Query Reverse Engineering. *Master's Thesis*, Tecnico Lisboa, Nov 2019. `web.ist.utl.pt/ist181151/81151-pedro-silva_dissertacao.pdf`

[6] R. DeMillo, R. Lipton and F. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4), 1978.

[7] K. Emani, K. Ramachandra, S. Bhattacharya and S. Sudarshan. Extracting Equivalent SQL from Imperative Code in Database Applications. In *Proc. of ACM SIGMOD Conf.*, 2016.

[8] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proc. of ACM POPL Conf.*, 2011.

[9] S. Gupta, S. Purandare, and K. Ramachandra. "Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates". In *Proc. of ACM SIGMOD Conf.*, 2020.

[10] R. Guravannavar, S. Sudarshan, A. Diwan and Ch. Babu. Which sort orders are interesting?. *The VLDB Journal* 21, 2012.

[11] D. Kalashnikov, L. Lakshmanan and D. Srivastava. FastQRE: Fast Query Reverse Engineering. In *Proc. of ACM SIGMOD Conf.*, 2018.

[12] G. Karvounarakis, Z. Ives and V. Tannen. Querying Data Provenance. In *Proc. of ACM SIGMOD Conf.*, 2010.

[13] K. Khurana and J. Haritsa. UNMASQUE: A Hidden SQL Query Extractor. *PVLDB*, 13(12), 2020.

[14] W. Kim. On Optimizing an SQL-like Nested Query *ACM TODS*, 7(3), 1982.

[15] H. Li, C. Chan and D. Maier. Query from Examples: An Iterative, Data-Driven Approach to Query Construction. *The VLDB Journal*, 8(13), 2015.

[16] K. Panev and S. Michel. Reverse Engineering Top-k Database Queries with PALEO. In *Proc. of EDBT Conf.*, 2016.

[17] H. Pirahesh, J. Hellerstein and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *Proc. of SIGMOD Conf.*,1992.

[18] K. Ramachandra, K. Park, K. Emani, A. Halverson, C. Galindo-Legaria and C. Cunningham. Froid: Optimization of Imperative Programs in a Relational Database. *PVLDB*, 11(4), 2017.

[19] S. Shah, S. Sudarshan, S. Kajbaje, S. Patidar, B. Gupta and D. Vira. Generating test data for killing SQL mutants: A constraint-based approach. In *Proc. of IEEE ICDE Conf.*, 2011.

[20] J. Shen and M. Rinard. Using Active Learning to Synthesize Models of Applications that Access Databases. In *Proc. of ACM PLDI Conf.*, 2019.

[21] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding and L. Novik. Discovering Queries based on Example Tuples. In *Proc. of ACM SIGMOD Conf.*, 2014.

[22] W. Tan, M. Zhang, H. Elmeleegy and D. Srivastava. Reverse Engineering Aggregation Queries. *PVLDB*, 10(11), 2017.

[23] W. Tan, M. Zhang, H. Elmeleegy and D. Srivastava. REGAL+: Reverse Engineering SPJA Queries. *PVLDB*, 11(12), 2018.

[24] Q. Tran, C. Chan and S. Parthasarathy. Query by Output. *Technical Report, National Univ. of Singapore*, 2009.

[25] Q. Tran, C. Chan and S. Parthasarathy. Query Reverse Engineering. *The VLDB Journal*, 23(5), 2014.

[26] J. Tuya, M. Cabal and C. Riva. Full predicate coverage for testing SQL database queries. *Software Testing Verification and Reliability*, 20(3), 2010.

[27] C. Wang, A.Cheung, R. Bodik. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proc. of ACM PLDI Conf.*, 2017.

[28] W. Wong, B. Kao, D. Cheung, R. Li and S. Yiu. Secure query processing with data interoperability in a cloud database environment. In *Proc. of ACM SIGMOD Conf.*, 2014.

[29] M. Zhang, H. Elmeleegy, C. Procopiuc, and D. Srivastava. Reverse Engineering Complex Join Queries. In *Proc. of ACM SIGMOD Conf.*, 2013.

[30] S. Zhang and Y. Sun. Automatically Synthesizing SQL Queries from Input-Output Examples. In *Proc. of IEEE ASE*, 2013.

[31] `www.bleepingcomputer.com/news/security/massive-wave-of-mongodb-ransom-atta` `makes-26-000-new-victims/`

[32] `www.cybersecurity-insiders.com/ransomware-hits-mysql-servers/`

[33] `www.hibernate.org`

[34] `www.imperva.com/blog/database-attacks-sql-obfuscation/`

[35] Jailer: `www.jailer.sourceforge.net/home.htm`

[36] IMDB dataset. `https://www.imdb.com/interfaces/`

[37] JOB. `https://github.com/gregrahn/join-order-benchmark`

[38] `https://github.com/xaviershay/enki`

[39] `https://projects.ow2.org/view/rubis`

[40] `docs.microsoft.com/en-us/sysinternals/downloads/strings`

[41] `https://www.selenium.dev/`

[42] `www.microsoft.com/en-in/sql-server`

[43] `www.mysql.com/`

[44] `www.postgresql.org`

[45] `https://rubyonrails.org/`

[46] `www.red-gate.com/blog/database-devops/database-subsetting-wed-love-hear`

[47] `www.softwareheritage.org/mission/software-is-fragile`

[48] `www.sql-shield.com`

[49] Condensor: `www.tonic.ai/post/condenser-a-database-subsetting-tool/`

[50] `www.tpc.org`

[51] UCI datasets. `https://archive.ics.uci.edu/ml/datasets.php`

[52] Wilos: an orchestration process software. `www.openhub.net/p/6390`

[53] `stackify.com/view-sql-with-prefix/`

[54] `https://dsl.cds.iisc.ac.in/projects/HIDDEN/unmasque.mp4`

Appendix

TPC-H Based Queries

Q_1

Select l_returnflag, l_linestatus, sum(l_quantity) **as** sum_qty, sum(l_extendedprice) **as** sum_base_price, sum(l_extendedprice * (1 - l_discount)) **as** sum_disc_price, sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) **as** sum_charge, avg(l_quantity) **as** avg_qty, avg(l_extendedprice) **as** avg_price, avg(l_discount) **as** avg_disc, count(*) **as** count_order
From lineitem
Where l_shipdate $<=$ date '1998-12-01' - interval '71 days'
Group By l_returnflag, l_linestatus
Order by l_returnflag, l_linestatus;

Q_2

Select s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone, s_comment
From part, supplier, partsupp, nation, region
Where p_partkey = ps_partkey **and** s_suppkey = ps_suppkey **and** p_size = 38 **and** p_type **like** '%TIN' **and** s_nationkey = n_nationkey **and** n_regionkey = r_regionkey **and** r_name = 'MIDDLE EAST'
Order by s_acctbal **desc**, n_name, s_name, p_partkey
Limit 100;

Q_3

Select l_orderkey, sum(l_extendedprice * (1 - l_discount)) **as** revenue, o_orderdate, o_shippriority
From customer, orders, lineitem
Where c_mktsegment = 'BUILDING' **and** c_custkey = o_custkey **and** l_orderkey = o_orderkey **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;

Q_4

Select o_orderdate, o_orderpriority, count(*) **as** order_count
From orders
Where o_orderdate $>=$ date '1997-07-01' **and** o_orderdate $<$ date '1997-07-01' + interval '3' month
Group By o_orderdate, o_orderpriority
Order by o_orderpriority
Limit 10;

Q_5

Select n_name, sum(l_extendedprice * (1 - l_discount)) **as** revenue
From customer, orders, lineitem, supplier, nation, region
Where c_custkey = o_custkey **and** l_orderkey = o_orderkey **and** l_suppkey = s_suppkey **and** c_nationkey = s_nationkey **and** s_nationkey = n_nationkey **and** n_regionkey = r_regionkey **and** r_name = 'MIDDLE EAST' **and** o_orderdate $>=$ date '1994-01-01' **and** o_orderdate $<$ date

'1994-01-01' + interval '1' year
Group By n_name
Order by revenue **desc**
Limit 100;

Q_6

Select l_shipmode, sum(l_extendedprice * l_discount) **as** revenue
From lineitem
Where l_shipdate $>=$ date '1994-01-01' **and** l_shipdate $<$ date '1994-01-01' + interval '1' year **and** l_quantity $<$ 24
Group By l_shipmode
Limit 100;

Q_{10}

Select c_name, sum(l_extendedprice * (1 - l_discount)) **as** revenue, c_acctbal, n_name, c_address, c_phone, c_comment
From customer, orders, lineitem, nation
Where c_custkey = o_custkey **and** l_orderkey = o_orderkey **and** o_orderdate $\geq$ date '1994-01-01' **and** o_orderdate $<$ date '1994-01-01' + interval '3' month **and** l_returnflag = 'R' **and** c_nationkey = n_nationkey
Group By c_name, c_acctbal, c_phone, n_name, c_address, c_comment
Order by revenue **desc**
Limit 20;

Q_{11}

Select ps_COMMENT, sum(ps_supplycost * ps_availqty) **as** value
From partsupp, supplier, nation
Where ps_suppkey = s_suppkey **and** s_nationkey = n_nationkey **and** n_name = 'ARGENTINA'
Group By ps_COMMENT
Order by value **desc**
Limit 100;

Q_{16}

Select p_brand, p_type, p_size, count(ps_suppkey) **as** supplier_cnt
From partsupp, part
Where p_partkey = ps_partkey **and** p_brand = 'Brand#45' **and** p_type **Like** 'SMALL PLATED%' **and** p_size $>=$ 4
Group By p_brand, p_type, p_size
Order by supplier_cnt **desc**, p_brand, p_type, p_size;

Q_{17}
Select AVG(l_extendedprice) as avgTOTAL
From lineitem, part
Where p_partkey = l_partkey **and** p_brand = 'Brand#52' **and** p_container = 'LG CAN' ;

Q_{18}

Select c_name, o_orderdate, o_totalprice, sum(l_quantity)
From customer, orders, lineitem
Where c_phone **Like** '27-_%' **and** c_custkey = o_custkey **and** o_orderkey = l_orderkey
Group By c_name, o_orderdate, o_totalprice
Order by o_orderdate, o_totalprice **desc**
Limit 100;

Q_{21}

Select s_name, count(*) **as** numwait
From supplier, lineitem l1, orders, nation
Where s_suppkey = l1.l_suppkey **and** o_orderkey = l1.l_orderkey **and** o_orderstatus = 'F' **and** s_nationkey = n_nationkey **and** n_name = 'GERMANY'
Group By s_name
Order by numwait **desc**, s_name
Limit 100;

TPC-DS Based Queries

Q_3

Select dt.d_year,item.i_brand_id as brand_id,item.i_brand as brand,sum(ss_sales_price) as sum_agg
From date_dim dt,store_sales,item
Where dt.d_date_sk = store_sales.ss_sold_date_sk **and** store_sales.ss_item_sk = item.i_item_sk
and item.i_manufact_id = 816 **and** dt.d_moy=11
Group By dt.d_year,item.i_brand,item.i_brand_id
Order by dt.d_year,sum_agg **desc**,brand_id
Limit 100 ;

Q_{37}

Select i_item_id,i_item_desc,i_current_price
From item, inventory, date_dim, catalog_sales
Where i_current_price **between** 45 **and** 45 + 30 **and** inv_item_sk = i_item_sk **and**
d_date_sk=inv_date_sk **and** d_date **between** date '1999-02-21' **and** date '1999-04-23' **and**
i_manufact_id **between** 707 **and** 1000 **and** inv_quantity_on_hand **between** 100 **and** 500 **and**
cs_item_sk = i_item_sk
Group By i_item_id,i_item_desc,i_current_price
Order by i_item_id
Limit 100 ;

Q_{42}

Select dt.d_year,item.i_category_id,item.i_category, sum(ss_ext_sales_price)
From date_dim dt,store_sales,item
Where dt.d_date_sk = store_sales.ss_sold_date_sk **and** store_sales.ss_item_sk = item.i_item_sk
and item.i_manager_id = 1 **and** dt.d_moy=11 **and** dt.d_year=2002
Group By dt.d_year,item.i_category_id,item.i_category
Order by sum(ss_ext_sales_price) **desc**,dt.d_year, item.i_category_id,item.i_category
Limit 100 ;

Q_{52}

Select dt.d_year,item.i_brand_id as brand_id,item.i_brand as brand,sum(ss_ext_sales_price) as
ext_price
From date_dim dt ,store_sales ,item
Where dt.d_date_sk = store_sales.ss_sold_date_sk **and** store_sales.ss_item_sk = item.i_item_sk
and item.i_manager_id = 1 **and** dt.d_moy=12 **and** dt.d_year=2002
Group By dt.d_year,item.i_brand,item.i_brand_id
Order by dt.d_year,ext_price **desc**,brand_id
Limit 100 ;

Q_{55}

Select item.i_brand_id as brand_id,item.i_brand as brand,sum(ss_ext_sales_price) as ext_price
From date_dim dt ,store_sales ,item
Where dt.d_date_sk = store_sales.ss_sold_date_sk **and** store_sales.ss_item_sk = item.i_item_sk
and item.i_manager_id = 1 **and** dt.d_moy=12 **and** dt.d_year=2002

Group By dt.d_year,item.i_brand,item.i_brand_id
Order by ext_price **desc**,brand_id
Limit 100 ;

Q_{82}

Select i_item_id,i_item_desc,i_current_price
From item, inventory, date_dim, store_sales
Where i_current_price **between** 45 **and** 45 + 30 **and** inv_item_sk = i_item_sk **and** d_date_sk=inv_date_sk **and** d_date **between** date '1999-07-09' **and** date '1999-09-09' **and** i_manufact_id **between** 169 **and** 639 **and** inv_quantity_on_hand **between** 100 **and** 500 **and** ss_item_sk = i_item_sk
Group By i_item_id,i_item_desc,i_current_price
Order by i_item_id
Limit 100 ;

Q_{96}

Select count(*)
From store_sales,household_demographics,time_dim, store
Where ss_sold_time_sk = time_dim.t_time_sk **and** ss_hdemo_sk = household_demographics.hd_demo_sk **and** ss_store_sk = s_store_sk **and** time_dim.t_hour = 8 **and** time_dim.t_minute $\geq$ 30 **and** household_demographics.hd_dep_count = 3 **and** store.s_store_name = 'ese'
Order by count(*)
Limit 100;

JOB Based Queries

Q_{7b}

```
SELECT MIN(n.name) AS of_person, MIN(t.title) AS biography_movie
FROM aka_name AS an,
cast_info AS ci,
info_type AS it,
link_type AS lt,
movie_link AS ml,
name AS n,
person_info AS pi,
title AS t
WHERE an.name LIKE '%a% '
AND it.info = 'mini biography '
AND lt.link = 'features '
AND n.name_pcode_cf LIKE 'D% '
AND n.gender= 'm '
AND pi.note = 'Volker Boehm '
AND t.production_year BETWEEN 1980 AND 1984
AND n.id = an.person_id
AND n.id = pi.person_id
AND ci.person_id = n.id
AND t.id = ci.movie_id
AND ml.linked_movie_id = t.id
AND lt.id = ml.link_type_id
AND it.id = pi.info_type_id
AND pi.person_id = an.person_id
AND pi.person_id = ci.person_id
AND an.person_id = ci.person_id
AND ci.movie_id = ml.linked_movie_id;
```

Q_{8a}

```
SELECT MIN(an1.name) AS actress_pseudonym, MIN(t.title) AS japanese_movie_dubbed
FROM aka_name AS an1,
cast_info AS ci,
company_name AS cn,
movie_companies AS mc,
name AS n1,
role_type AS rt,
title AS t
WHERE ci.note = '(voice: English version) '
AND cn.country_code = '[jp] '
AND mc.note LIKE '%(Japan)% '
AND n1.name LIKE '%Yo% '
AND rt.role = 'actress '
AND an1.person_id = n1.id
AND n1.id = ci.person_id
AND ci.movie_id = t.id
AND t.id = mc.movie_id
AND mc.company_id = cn.id
AND ci.role_id = rt.id
AND an1.person_id = ci.person_id
AND ci.movie_id = mc.movie_id;
```

Q_{9b}

```
SELECT MIN(an.name) AS alternative_name, MIN(chn.name) AS voiced_character, MIN(n.name) AS
voicing_actress, MIN(t.title) AS american_movie
FROM aka_name AS an,
char_name AS chn,
cast_info AS ci,
company_name AS cn,
movie_companies AS mc,
name AS n,
role_type AS rt,
title AS t
WHERE ci.note = '(voice) '
AND cn.country_code = '[us] '
AND mc.note LIKE '%(200%)% '
AND n.gender = 'f '
AND n.name LIKE '%Angel% '
AND rt.role = 'actress '
AND t.production_year BETWEEN 2007 AND 2010
AND ci.movie_id = t.id
AND t.id = mc.movie_id
AND ci.movie_id = mc.movie_id
AND mc.company_id = cn.id
AND ci.role_id = rt.id
AND n.id = ci.person_id
AND chn.id = ci.person_role_id
AND an.person_id = n.id
AND an.person_id = ci.person_id;
```

Q_{10a}

```
SELECT MIN(chn.name) AS uncredited_voiced_character, MIN(t.title) AS russian_movie
FROM char_name AS chn,
cast_info AS ci,
company_name AS cn,
company_type AS ct,
movie_companies AS mc,
role_type AS rt,
title AS t
WHERE ci.note LIKE '%(uncredited)% '
AND cn.country_code = '[ru] '
AND rt.role = 'actor '
AND t.production_year > 2005
AND t.id = mc.movie_id
AND t.id = ci.movie_id
```

AND ci.movie_id = mc.movie_id
AND chn.id = ci.person_role_id
AND rt.id = ci.role_id
AND cn.id = mc.company_id
AND ct.id = mc.company_type_id;

Q_{11a}

SELECT MIN(cn.name) AS from_company, **MIN**(lt.link) AS movie_link_type, **MIN**(t.title) AS non_polish_sequel_movie
FROM company_name AS cn,
company_type AS ct,
keyword AS k,
link_type AS lt,
movie_companies AS mc,
movie_keyword AS mk,
movie_link AS ml,
title AS t
WHERE cn.country_code = '[pl] '
AND cn.name **LIKE** '%Film% '
AND ct.kind = 'production companies '
AND k.keyword = 'sequel '
AND lt.link **LIKE** '%follow% '
AND mc.note **IS NULL**
AND t.production_year **BETWEEN** 1950 AND 2000
AND lt.id = ml.link_type_id
AND ml.movie_id = t.id
AND t.id = mk.movie_id
AND mk.keyword_id = k.id
AND t.id = mc.movie_id
AND mc.company_type_id = ct.id
AND mc.company_id = cn.id
AND ml.movie_id = mk.movie_id
AND ml.movie_id = mc.movie_id
AND mk.movie_id = mc.movie_id;

Q_{15a}

SELECT MIN(mi.info) AS release_date, **MIN**(t.title) AS internet_movie
FROM aka_title AS at,
company_name AS cn,
company_type AS ct,
info_type AS it1,
keyword AS k,
movie_companies AS mc,
movie_info AS mi,
movie_keyword AS mk,
title AS t
WHERE cn.country_code = '[us] '
AND it1.info = 'release dates '

```
AND mc.note LIKE '%(worldwide)% '
AND mi.note LIKE '%internet% '
AND mi.info LIKE 'USA:% 200% '
AND t.production_year > 2000
AND t.id = at.movie_id
AND t.id = mi.movie_id
AND t.id = mk.movie_id
AND t.id = mc.movie_id
AND mk.movie_id = mi.movie_id
AND mk.movie_id = mc.movie_id
AND mk.movie_id = at.movie_id
AND mi.movie_id = mc.movie_id
AND mi.movie_id = at.movie_id
AND mc.movie_id = at.movie_id
AND k.id = mk.keyword_id
AND it1.id = mi.info_type_id
AND cn.id = mc.company_id
AND ct.id = mc.company_type_id;
```

Q_{17f}

```
SELECT MIN(n.name) AS member_in_charnamed_movie
FROM cast_info AS ci,
company_name AS cn,
keyword AS k,
movie_companies AS mc,
movie_keyword AS mk,
name AS n,
title AS t
WHERE k.keyword = 'character-name-in-title '
AND n.name LIKE '%B% '
AND n.id = ci.person_id
AND ci.movie_id = t.id
AND t.id = mk.movie_id
AND mk.keyword_id = k.id
AND t.id = mc.movie_id
AND mc.company_id = cn.id
AND ci.movie_id = mc.movie_id
AND ci.movie_id = mk.movie_id
AND mc.movie_id = mk.movie_id;
```

Q_{19a}

```
SELECT MIN(n.name) AS voicing_actress, MIN(t.title) AS voiced_movie
FROM aka_name AS an,
char_name AS chn,
cast_info AS ci,
company_name AS cn,
info_type AS it,
movie_companies AS mc,
movie_info AS mi,
name AS n,
role_type AS rt,
title AS t
WHERE ci.note = '(voice: Japanese version) '
AND cn.country_code = '[us] '
AND it.info = 'release dates '
AND mc.note IS NOT NULL
AND mc.note LIKE '%(worldwide)% '
AND mi.info IS NOT NULL
AND mi.info LIKE 'Japan:%200% '
AND n.gender = 'f '
AND n.name LIKE '%Ang% '
AND rt.role = 'actress '
AND t.production_year BETWEEN 2005 AND 2009
AND t.id = mi.movie_id
AND t.id = mc.movie_id
AND t.id = ci.movie_id
AND mc.movie_id = ci.movie_id
AND mc.movie_id = mi.movie_id
AND mi.movie_id = ci.movie_id
AND cn.id = mc.company_id
AND it.id = mi.info_type_id
AND n.id = ci.person_id
AND rt.id = ci.role_id
AND n.id = an.person_id
AND ci.person_id = an.person_id
AND chn.id = ci.person_role_id;
```

Q_{21c}

```
SELECT MIN(cn.name) AS company_name, MIN(lt.link) AS link_type, MIN(t.title) AS western_follow_up
FROM company_name AS cn,
company_type AS ct,
keyword AS k,
link_type AS lt,
movie_companies AS mc,
movie_info AS mi,
movie_keyword AS mk,
movie_link AS ml,
title AS t
WHERE cn.country_code != '[pl] '
AND cn.name LIKE '%Warner% '
AND ct.kind = 'production companies '
AND k.keyword = 'sequel '
AND lt.link LIKE '%follow% '
AND mc.note IS NULL
AND mi.info = 'Sweden '
AND t.production_year BETWEEN 1950 AND 2010
AND lt.id = ml.link_type_id
AND ml.movie_id = t.id
AND t.id = mk.movie_id
AND mk.keyword_id = k.id
AND t.id = mc.movie_id
AND mc.company_type_id = ct.id
AND mc.company_id = cn.id
AND mi.movie_id = t.id
AND ml.movie_id = mk.movie_id
AND ml.movie_id = mc.movie_id
AND mk.movie_id = mc.movie_id
AND ml.movie_id = mi.movie_id
AND mk.movie_id = mi.movie_id
AND mc.movie_id = mi.movie_id;
```

Q_{23c}

```
SELECT MIN(kt.kind) AS movie_kind, MIN(t.title) AS complete_us_internet_movie
FROM complete_cast AS cc,
comp_cast_type AS cct1,
company_name AS cn,
company_type AS ct,
info_type AS it1,
keyword AS k,
kind_type AS kt,
movie_companies AS mc,
movie_info AS mi,
movie_keyword AS mk,
title AS t
WHERE cct1.kind = 'complete+verified '
AND cn.country_code = '[us] '
AND it1.info = 'release dates '
AND kt.kind = 'movie '
AND mi.note LIKE '%internet% '
AND mi.info IS NOT NULL
AND mi.info LIKE 'USA:% 199% '
AND t.production_year > 1990
AND kt.id = t.kind_id
AND t.id = mi.movie_id
AND t.id = mk.movie_id
AND t.id = mc.movie_id
AND t.id = cc.movie_id
AND mk.movie_id = mi.movie_id
AND mk.movie_id = mc.movie_id
AND mk.movie_id = cc.movie_id
AND mi.movie_id = mc.movie_id
AND mi.movie_id = cc.movie_id
AND mc.movie_id = cc.movie_id
AND k.id = mk.keyword_id
AND it1.id = mi.info_type_id
AND cn.id = mc.company_id
```

AND ct.id = mc.company_type_id
AND cct1.id = cc.status_id;

Q_{24b}

SELECT MIN(chn.name) AS voiced_char_name, **MIN**(n.name) AS voicing_actress_name, **MIN**(t.title) AS kung_fu_panda
FROM aka_name AS an,
char_name AS chn,
cast_info AS ci,
company_name AS cn,
info_type AS it,
keyword AS k,
movie_companies AS mc,
movie_info AS mi,
movie_keyword AS mk,
name AS n,
role_type AS rt,
title AS t
WHERE ci.note = '(voice: Japanese version) '
AND cn.country_code = '[us] '
AND cn.name = 'DreamWorks Animation '
AND it.info = 'release dates '
AND k.keyword = 'martial-arts '
AND mi.info **IS NOT NULL**
AND mi.info **LIKE** 'USA:%201% '
AND n.gender = 'f '
AND n.name **LIKE** '%An% '
AND rt.role = 'actress '
AND t.production_year $>$ 2010
AND t.title **LIKE** 'Kung Fu Panda% '
AND t.id = mi.movie_id
AND t.id = mc.movie_id
AND t.id = ci.movie_id
AND t.id = mk.movie_id
AND mc.movie_id = ci.movie_id
AND mc.movie_id = mi.movie_id
AND mc.movie_id = mk.movie_id
AND mi.movie_id = ci.movie_id
AND mi.movie_id = mk.movie_id
AND ci.movie_id = mk.movie_id
AND cn.id = mc.company_id
AND it.id = mi.info_type_id
AND n.id = ci.person_id
AND rt.id = ci.role_id
AND n.id = an.person_id
AND ci.person_id = an.person_id
AND chn.id = ci.person_role_id
AND k.id = mk.keyword_id;