Unmasking Hidden SQL Queries

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Technology
IN
Faculty of Engineering

BY

Kapil Khurana

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

July, 2020

Declaration of Originality

I, **Kapil Khurana**, with SR No. **04-04-00-10-42-18-1-16205** hereby declare that the material presented in the thesis titled

Unmasking Hidden SQL Queries

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2018-2020**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Prof. Jayant R. Haritsa Advisor Signature

DEDICATED TO

My Parents
Mrs. Versha Khurana and the late Mr. Omprakash Khurana;

For their endless love, support and encouragement

Acknowledgements

I would like to express my sincere gratitude to my project advisor, Prof. Jayant R. Haritsa for giving me an opportunity to work on this project with him. I am thankful to him for his valuable guidance and continuous support at each and every step. When the going gets tough, the tough get going. That is the kind of spirit he has always worked with and imparted to me.

I would also like to thank the Department of Computer Science and Automation and Indian Institute of Science for providing excellent study environment and research facilities. The learning experience has been really great here. I would like to thank all IISc staff for the role each of them played throughout my journey at IISc.

I would like to thank my lab mates for their valuable inputs and constructive criticism and my friends for supporting me during critical times. Finally, I would like to thank my parents who has always supported me indefinitely.

Abstract

We investigate here the query reverse-engineering problem of unmasking SQL queries hidden within database applications, a problem with use-cases ranging from legacy code to server security. As a first step in addressing this challenge, we present UNMASQUE, an extraction algorithm that is capable of identifying a substantive class of complex hidden queries. A special feature of our design is that the extraction is non-invasive w.r.t. the application, examining only the results obtained from its executions on databases derived with a combination of data mutation and data generation techniques.

Further, potent optimizations, such as database size reduction to a few rows, are incorporated to minimize the extraction overheads. A detailed evaluation over benchmark databases demonstrates that UNMASQUE is capable of correctly and efficiently extracting a broad spectrum of hidden queries. We also show UNMASQUE's capability to convert imperative code into equivalent SQL queries for some famous blogging applications.

Publications based on this Thesis

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past decade, *query reverse-engineering* (QRE) has evinced considerable interest from both the database and programming language communities (e.g. [14, 11, 10, 9, 5, 3, 2, 7, 4, 13]). 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.

Impressive progress has been made on addressing the QRE problem, with potent tools such as Talos[11], Regal[10] and Scythe[13] having been developed over the years. Notwithstanding, there are intrinsic challenges underlying the problem framework: First, the choice of candidate query is organically dependent on the specific ($\mathcal{D}_i$,$\mathcal{R}_i$) instance provided by the user, and can vary hugely based on this initial sample. As an extreme case in point, if the result has only a single row, the generated candidates are likely to be trivial queries, although the ideal answer may be an aggregation query. Second, given the inherently exponential search space of alternatives, identifying and selecting among the candidates is not easily amenable to efficient processing. Third, the precise values of filter predicates, as well as advanced SQL constructs (e.g. Limit, Like, UDFs), are fundamentally impossible to deduce since the candidate query is constructed solely from the instance.

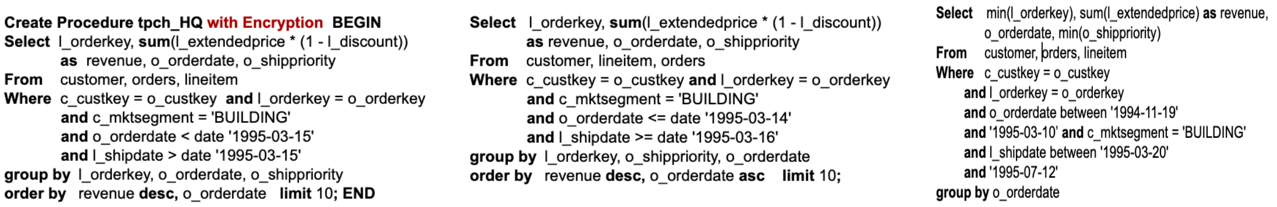
In this report, we consider a variant of the QRE problem, wherein a *ground-truth* query is additionally available, but in a hidden form that is not easily accessible. For example, the original query may be explicitly hidden in a black-box application executable. Moreover, encryption or obfuscation may have been additionally incorporated to further protect the application logic. An alternative scenario is that the application is visible but effectively opaque because it is comprised of hard-to-comprehend SQL (such as those arising from machine-generated object-relational mappings), or poorly documented imperative code that is not easily decipherable. Such "hidden-executable" situations could arise in the context of legacy code, where the original source has been lost or misplaced over time (prominent instances of such losses are recounted in [22]), or when third-party proprietary tools are part of the workflow, or if the software has been 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 SQL query $\mathcal{Q}_H$, and a database instance $\mathcal{D}_i$ on which $\mathcal{Q}_H$ produces a populated result $\mathcal{R}_i$, unmask $\mathcal{Q}_H$ to reveal the original query.*

We leverage the presence of the hidden ground-truth to deliver a variety of advantages:

- The outcome now becomes independent of the initial ($\mathcal{D}_i$,$\mathcal{R}_i$) instance.
- Since the application can be invoked repeatedly on different databases, efficient and focused mechanisms can be designed to precisely identify the hidden query.
- It allows for capturing difficult SQL constructs (we show here how Like, Limit, Having and scalar UDFs can be extracted).
- As a collateral benefit, the unmasked query can serve as a definitive seed input to database usability tools like Talos[11] which create an array of instance-equivalent queries.
- New use-cases become feasible – for instance, a security agency may wish to ascertain offline the real intent of encrypted queries that were refused entry due to concerns about their origins.

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 later, this is not a viable approach, forcing us to design the extraction procedures from scratch. Our experience in this effort is that HQE proves to be a challenging research problem due to factors such as (a) acute dependencies between the various clauses of the hidden query, (b) possibility of schematic renaming, (c) result compression due to aggregration functions, and (d) presence of UDFs.



```
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) **Extracted Query ($\mathcal{Q}_E$)**

```
Select    min(l_orderkey), sum(l_extendedprice) as revenue,
          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 c_mktsegment = 'BUILDING'
      and l_shipdate between '1995-03-20'
      and '1995-07-12'
group by o_orderdate
```

(c) **Sample Regal Query**

Figure 1.1: Hidden Query Extraction Example (TPC-H Q3)

1.1 UNMASQUE Algorithm

We take a first step towards addressing the HQE problem here by presenting UNMASQUE[1], an algorithm that uses a judicious combination of *database mutation* and synthetic *database generation* to identify the hidden query $\mathcal{Q}_H$. The extraction is completely *non-invasive* wrt the application code, examining only the results obtained from its executions on carefully constructed databases. As a result, platform-independence is achieved wrt the underlying database engine.

Currently, UNMASQUE is capable of extracting a substantive class of SPJGHAOL[2] queries. As an exemplar, consider $\mathcal{Q}_H$ in Figure 1.1a, which encrypts TPC-H [25] query Q3 in a stored procedure, and features most of these clauses. Our extracted equivalent, $\mathcal{Q}_E$, is shown in Figure 1.1b, clearly capturing all *semantic* aspects of the original query, including the scalar revenue UDF. Only syntactic differences, such as a different grouping column order, remain in the extraction.

As a reference point, the candidate query produced by Regal+[10] for this scenario is shown in Figure 1.1c. While the query tables and joins are detected correctly, there are significant discrepancies in the filters, grouping columns and aggregation functions. Moreover, the query is produced after removing limit, order and UDF clauses and converting character and date type columns to integers to suite their environment. Finally, producing even this limited outcome took considerable time and resources.

1.2 Extraction Workflow

UNMASQUE operates according to the pipeline shown in Figure 1.2, where it unmasks the hidden query elements in a structured manner. It starts with the FROM clause, continues on to the JOIN and FILTER predicates, follows up with the PROJECTION and GROUP BY+AGGREGATION

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

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

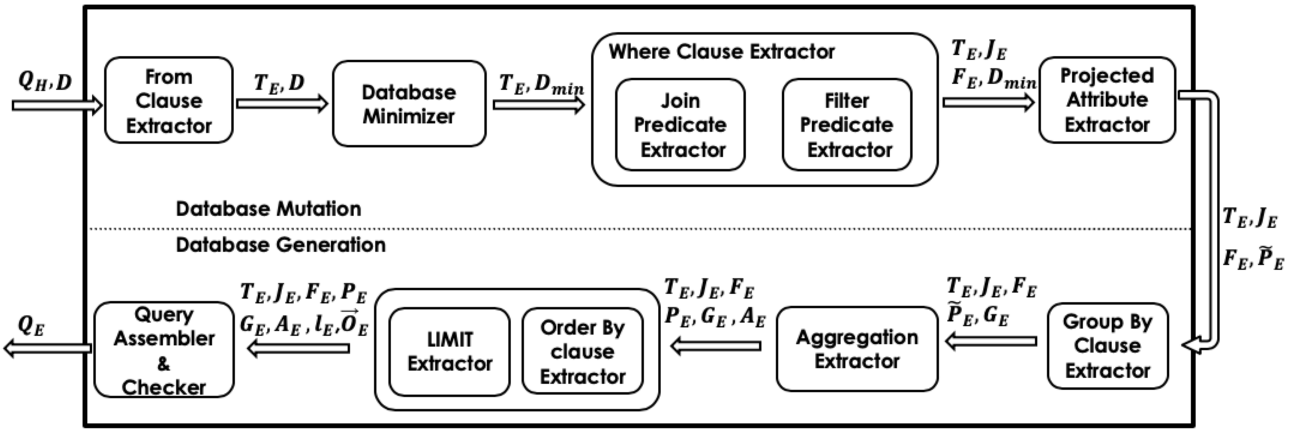


Figure 1.2: UNMASQUE Architecture

columns, and concludes with the ORDER BY and LIMIT functions (as explained in Section 6.2, a different pipeline structure is required to extract HAVING clause). The initial elements are extracted using *database mutation* strategies, whereas the subsequent ones are extracted leveraging *database generation* techniques. Further, while some of the elements are relatively easy to extract (e.g. FROM), there are others (e.g. GROUP BY) that require carefully crafted methods for unambiguous identification. The final component in the pipeline is the QUERY ASSEMBLER which puts together the different elements of $\mathcal{Q}_E$ and performs canonification to ensure a standard output format.

1.3 Extraction Efficiency

To cater to extraction efficiency concerns, UNMASQUE incorporates a variety of optimizations. In particular, it solves a conceptual problem of independent interest: *Given a database instance $\mathcal{D}$ on which a hidden query $\mathcal{Q}_H$ produces a populated result $\mathcal{R}$, identify the smallest subset $\mathcal{D}_{min}$ of $\mathcal{D}$ such that the result of $\mathcal{Q}_H$ 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. [6]). However, due to the *hidden* nature of $\mathcal{Q}_H$, these approaches are no longer 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 has been identified, as shown in Figure 1.2. And the reduction is always to the extent that the subsequent SPJ extraction is carried out on *miniscule* databases containing just a handful of rows. In an analogous fashion, the synthetic databases created for the GAOL extraction are also carefully designed to be very thinly populated. Overall, these reductions make the post-minimization processing to be essentially independent of database size.

1.4 Performance Evaluation

We have evaluated UNMASQUE's behavior on a suite of complex decision-support queries, and on imperative code sourced from blogging tools. The performance results of these experiments, conducted on a vanilla PostgreSQL [20] platform, indicate that UNMASQUE precisely identifies the hidden queries in our workloads in a timely manner. As a case in point, the extraction of the example Q3 on a 100 GB TPC-H database was completed within *10 minutes*. This performance is especially attractive considering that a native execution of Q3 takes around 5 minutes on the same platform.

1.5 Organization

The rest of the report is organized as follows: In Chapter 2, a precise description of the HQE problem is provided, along with the notations. The following chapters – Chapter 3 and 4 – 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 Chapter 5. Extraction of the Having clause and other extensions are discussed in Chapter 6. Chapter 7 summarizes some theoretical results about HQE. Finally, our conclusions and future research avenues are summarized in Chapter 8.

Chapter 2

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* [27], 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* [17]). 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[23] 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 in the database for extraction purposes.

2.1 Extractable Query Class

The QRE literature has primarily focused on constructing generic SPJGA queries that do not feature non-equi-joins, nesting, disjunctions or UDFs. We share some of the restrictions but have been able to extend the query extraction scope to include HOL constructs, as well as

Symbol	Meaning	Symbol	Meaning(wrt query $\mathcal{Q}_E$)
$\mathcal{A}$	Application	T_E	Set of tables in query
$\mathcal{F}$	Application Executable	C_E	Set of columns in T_E
D	Initial Database	JG_E	Join graph
R	Result of $\mathcal{F}$ on D	J_E	Set of join predicates
T	Set of all tables in D	F_E	Set of filter predicates
$\mathcal{Q}_H$	Hidden Query	P_E	Set of native projections with mapped result columns
$\mathcal{Q}_E$	Extracted Query	A_E	Set of aggregations with mapped result columns
D_{min}	Reduced Database	G_E	Set of group by columns
SG	Schema Graph of database	H_E	Set of having predicates
		$\overrightarrow{\mathcal{O}_E}$	Sequence of ordering result columns
		l_E	limit value

Table 2.1: Notations

simple scalar UDFs. Further, we expect join graph to be a subgraph of schema graph. There are additional mild constraints on some of the constructs – for instance, the LIMIT value must be at least 3, there are no filters on key attributes – and they are mentioned in the relevant locations in the following chapters. We hereafter refer to this class of supported queries as *Extractable Query Class* (EQC). Our subsequent description of UNMASQUE on EQC uses the sample TPCH Query 3 of the Introduction (Figure 1.1a) as the running example.

For ease of exposition and due to space limitations, we initially present UNMASQUE for SPJGAOL queries, and defer the HAVING clause to Section 6.2. 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 2.1. To highlight its black-box nature, the application executable is denoted by $\mathcal{F}$, while $\overrightarrow{\mathcal{O}_E}$ has a vector symbol to indicate that the ordering columns form a *sequence*.

2.2 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 $\mathcal{Q}_E$:

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

and sequentially identify each of the constituent elements, as per the pipeline shown in Figure 1.2.

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.

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

Chapter 3

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.

3.1 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*. Then, $\mathcal{F}$ is executed on this mutated schema and we check whether it throws an error – if yes, t is part of the query; 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. With $Q3$, the procedure results in

$T_E = \{$***customer, lineitem, orders***$\}$.

3.2 Database Minimization

For enterprise database applications, it is likely that D is huge, and therefore repeatedly executing $\mathcal{F}$ 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 D and an executable $\mathcal{F}$ producing a populated result on D, derive a reduced database instance D_{min} from D such that removing any row of any table in T_E results in an empty result.

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

Lemma 3.1: *For the EQC^{-H}, there always exists a D_{min} wherein each table in T_E contains only a **single** row.*

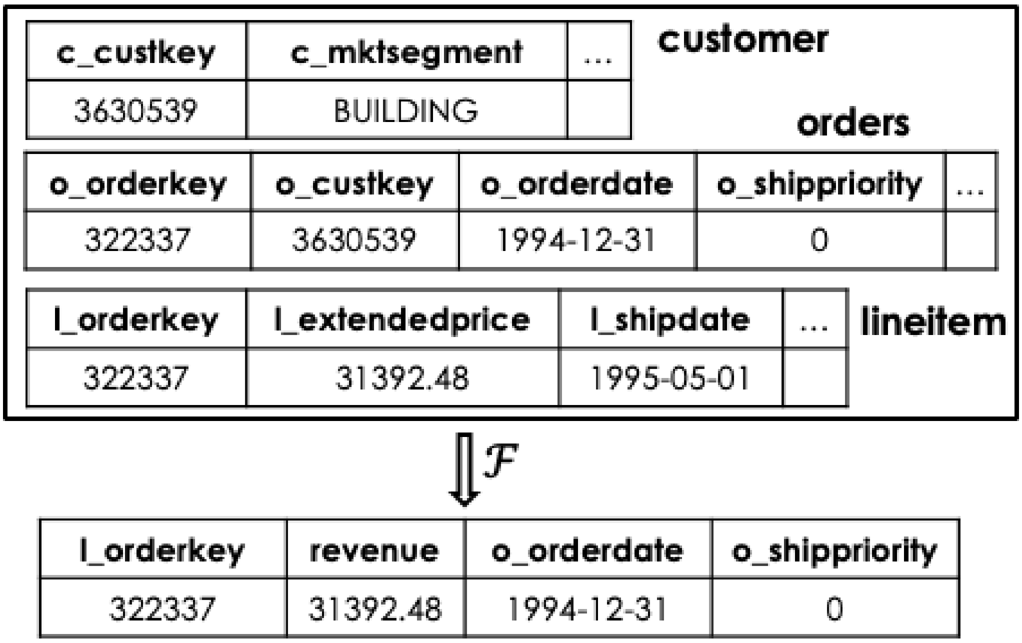
Proof: Firstly, since the final result is known to be populated, the intermediate result obtained after the evaluation of the SPJ core of the query is also guaranteed to be non-empty. This is because the subsequent GAOL elements only perform computations on the intermediate result but do not add to it. 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.

Reducing D to D^1

At first glance, it might appear trivial to identify a D^1– simply pick any row from the R obtained on D and compute its provenance using the well-established techniques in the literature (e.g. [6]) – 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 $\mathcal{F}$ 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, note that after each halving, $\mathcal{F}$ is executed to determine which half to retain, and therefore we would like to minimize the time taken by these executions. Accordingly, we choose a policy of always halving the *currently largest* table in the set. This is because this policy can be shown to



customer

c_custkey	c_mktsegment	...
3630539	BUILDING	

orders

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

lineitem

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

$\Downarrow \mathcal{F}$

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

Figure 3.1: D^1 for Q3

require, in expectation, the least amount of data processing 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 3.1.

3.3 Join Predicates

To extract the join predicates J_E of $\mathcal{Q}_H$, we start with SG, the original schema graph of the database. Note that, the nodes in SG are key columns (and not tables which is usually the case with the term, schema graph) and each edge (u, v) denotes an equi-join predicate $u = v$. From SG, we create an (undirected) induced subgraph whose vertices are the key columns in T_E, and edges are the possible join edges between these columns. In the case of composite keys, each column within the key is treated as a separate node.

After that, each connected component in the subgraph is converted to a corresponding *cycle graph*, hereafter referred to as a cycle, with the same set of vertices. Note that the elementary graph with two nodes and and an edge connecting them is also considered to be a cycle. The motivation for this graph conversion step is the following: Checking for the presence of a connected component in the query join graph JG_E, is equivalent to checking the presence of the corresponding cycle. Therefore the collection of all the cycles put together is referred to as *candidate join-graph*, or CJG_E.

We now individually check for the presence of each CJG_E cycle in JG_E, using the iterative procedure shown in Algorithm 1. The check is done in the following three steps: (i) Using the Cut procedure, remove a pair of edges from a candidate cycle CC; this partitions CC into two connected components; these new components are converted into cycles (C_1 and C_2) by adding the missing edge; (ii) Negate in D^1 all the values in the columns corresponding to the vertices in C_1, using the Negate procedure; (iii) Run $\mathcal{F}$ on this mutated database – if the result is empty, we conclude the edges are present in JG_E and the edges are returned to the parent cycle CC; otherwise, C_1 and C_2 are included as fresh candidates in CJG_E. If a candidate cycle has reduced to a single edge, then the check is carried out only with the Negation step using one of the two vertices.

In the above procedure, the motivation behind removing a *pair* of edges is the observation that for JG_E to not contain a cycle CC, at least *two* edges of CC should be absent from JG_E. The reason is that, in the above context, if an edge is removed from a cycle, the resultant graph is still equivalent to the cycle due to transitivity property of inner-equi join over columns. Further, the algorithm is bound to terminate because in each iteration, a cycle is either removed or partitioned into smaller cycles.

With regard to $Q3$, CJG_E contains only two connected components – specifically, $(l\_orderkey, o\_order$ 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, for Q3, the join predicates turn out to be:

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

Lemma 3.2: For a hidden query $\mathcal{Q}_H \in$ EQC, UNMASQUE correctly extracts JG_E, or equivalently, J_E.

Proof: It is easy see that when there is only one edge in the cycle, it will be correctly extracted as the output after removing it 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 UNMASQUE fails to extract it (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 UNMASQUE extracts it (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 joins (u, v) also belongs to the join graph as it can be inferred by other edges of cycle C, a contradiction. □

Algorithm 1: Extracting Join Graph JG_E

```
$CJG_E \leftarrow$ Candidate Cycles, $JG_E \leftarrow \phi$
**while** *There is at least one cycle in* $CJG_E$ **do**
    $CC \leftarrow$ Any candidate cycle from $CJG_E$
    **if** $CC$ *contains a single edge* $(v_1, v_2)$ **then**
        $D^1_{mut} \leftarrow$ Negate($D^1$, $\{v_1\}$)
        **If** $\mathcal{F}$ $(D^1_{mut}) = \phi$ **then** $JG_E \leftarrow JG_E \cup CC$
        $CJG_E \leftarrow CJG_E$ / $CC$
    **else**
        **foreach** *pair of edges* $(e_1, e_2) \in CC$ **do**
            $C_1, C_2 = Cut(CC, e_1, e_2)$
            $D^1_{mut} \leftarrow$ Negate($D^1$, $C_1$)
            **if** $\mathcal{F}$ $(D^1_{mut}) = \phi$ **then**
                Add $e_1$, and $e_2$ back to $CC$
            **else**
                $CJG_E \leftarrow CJG_E \cup C_1 \cup C_2$
                break     //Go to the start of while loop
            **end**
        **end**
        $JG_E \leftarrow JG_E \cup CC$;  $CJG_E \leftarrow CJG_E$ / $CC$
    **end**
**end**
```

3.4 Filter Predicates

We start by assuming that all columns in C_E (set of columns in T_E) are potential candidates for the filter predicates F_E in $\mathcal{Q}_H$. 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 $\mathcal{F}$ on these mutations. The procedure for general numeric and textual attributes is, however, more involved, as explained below.

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 3.1: Filter Predicate Cases

3.4.1 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 $\mathcal{F}$ 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 3.1.

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 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.

3.4.2 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.

3.4.3 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.

3.4.4 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"*. $\mathcal{F}$ 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 $\mathcal{F}$ 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 3.3: For a query in EQC, Algorithm 2 correctly identifies MQS for a filter predicate on character attribute.

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.

□

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 $\mathcal{F}\ (D^1_{mut}) = \phi$ 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 $\mathcal{F}\ (D^1_{mut}) = \phi$ then
            $MQS$.append('_'); $itr_{++}$
        else
            $rep\_str$.remove_char_at($itr$)
        end
    end
end
```

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 $\mathcal{F}$ 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 ≤ date '1995-03-14' ,**
l_shipdate ≥ date '1995-03-16' ,
c_mktsegment = 'BUILDING' }

3.5 Projections

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:

$$aA + bB + cAB + d = 0 \tag{3.1}$$

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.

3.5.1 Dependency List Identification

In this step, for each On, the set of database columns which affect its value is discovered via iterative column exploration and database mutation. Specifically, the value of each database column in C_E (the set of columns in T_E) is mutated in turn to see whether or not it affects the value of O. 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. To make it more concrete, if the value of column A in D^1 happen to be $\frac{-b}{c}$, then 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 value of column B in D^1 happen to be $\frac{-a}{c}$, then column A is blocked by column B. To address such boundary conditions, we perform a second iteration in case the dependency list contains less than two columns after first iteration. However, before the second iteration, the values in all the database columns are changed to new values keeping filter predicates in consideration. Now, if a column A was blocked before by another column B, it will no longer be blocked due to the change in the value in column B, and hence - it will be identified in the second iteration.

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

For $Q3$, the following dependency lists are obtained with the above procedure: *l_orderkey: [l_orderkey], o_orderdate: [l_orderkey], o_shippriority: [o_shippriority],* and *revenue: [l_extendedprice, l_discount].*

3.5.2 Function Identification

With reference to Equation 3.1, 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:

$$1.a + 2.b + 2.c + d = -1 \tag{3.2}$$

$$2.a + 1.b + 2.c + d = 0 \tag{3.3}$$

$$2.a + 3.b + 6.c + d = -4 \tag{3.4}$$

$$1.a + 4.b + 4.c + d = -3 \tag{3.5}$$

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:
$\widetilde{P_E}$ = **{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.

A closing note regarding the scope of scalar UDFs currently covered in UNMASQUE: Firstly, 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. Secondly, we can handle the CASE switch statements on categorical domains, such as those seen in TPCH Q12. Finally, ancillary functions such as *substring, casting, median, etc.* can also be extracted.

Chapter 4

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 chapter.

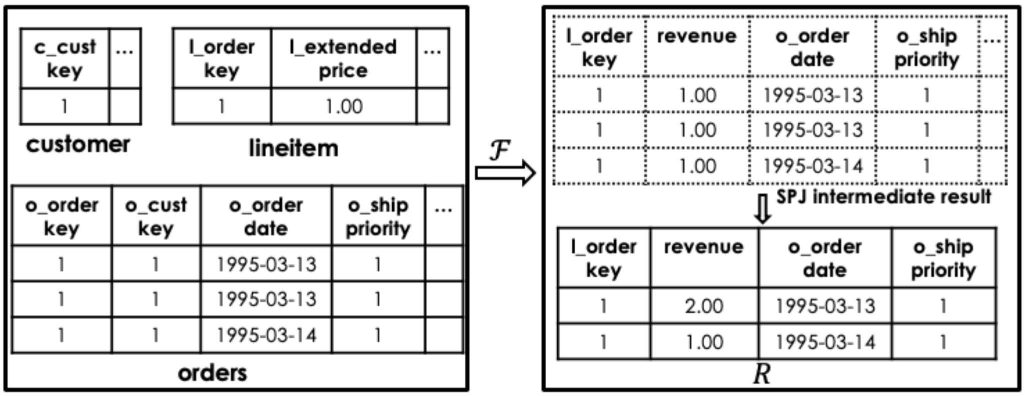
4.1 Group By Columns

For each column $t.A$ in C_E (the set of columns in T_E), we generate a database instance D_{gen} and analyze $\mathcal{F}$ (D_{gen}) for the existence of $t.A$ in G_E, the columns in the GROUP BY clause. However, we skip this check for columns with equality filter predicates (as determined in Mutation Pipeline) since their presence or absence in G_E makes no difference to the query output.

Assume for the moment that we have generated a D_{gen} such that the (invisible) *intermediate* result produced by the SPJ part of $\mathcal{Q}_H$ contains **3** rows satisfying the following condition: $t.A$ has a common value in exactly two rows, while all other columns have the same value in all three rows. Now, if the *final* 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 [8, 12].

Generating D_{gen}

We now explain how to produce the desired D_{gen} for checking the G_E membership of a generic column $t.A$. In our description, assigning $(p, q, r, ...)$ to $t.A$ means assigning value p in the first row, q in the second row, r in the third and so on. The database generation is performed differently for the following two disjoint cases related to the presence or absence of $t.A$ in the JG_E, the query join graph identified in Mutation Pipeline:



c_cust key	...
1	

customer

l_order key	l_extended price	...
1	1.00	

lineitem

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	

orders

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	

SPJ intermediate result

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

R

Figure 4.1: D_{gen} for Grouping on o_orderdate (Q3)

(Case 1) $t.A \notin JG_E$ In this case, 3 rows are generated for table t and only one row in each of the other tables in T_E. For column $t.A$, any two different values p and q that satisfy all associated filter predicates are assigned. If no filter exists, any two values from $t.A$'s domain are taken (e.g. $p = 1$ and $q = 2$ for numeric). After that, we assign (p, p, q) to $t.A$.

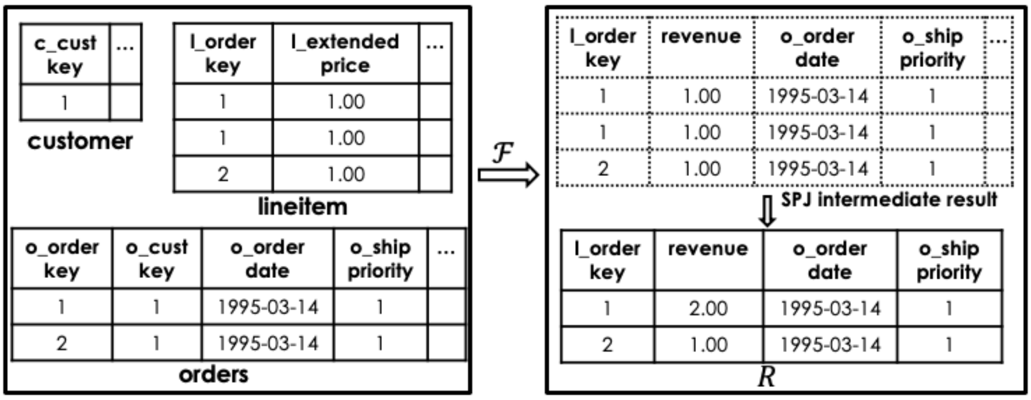
For all other columns in t, such as $t.X$, a single value r that satisfies its associated filter predicates (if any) is selected, and (r, r, r) is assigned to $t.X$. If there is no filter, any value from its domain (e.g. $r = 1$ for numeric) is assigned. Finally, if $t.X \in JG_E$, a fixed value of $r = 1$ is assigned (consistent with the assumption of integral keys). A similar assignment policy is used for all columns belonging to the remaining tables in T_E.

An example D_{gen} for checking the presence of o_orderdate in G_E is shown in Figure 4.1. 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. (We hasten to add that these intermediate results are shown just for illustrative purposes, but remain invisible to the UNMASQUE tool in its extraction process.)

(Case 2) $t.A \in JG_E$ In this case, 3 rows are generated for table t, 2 rows are generated 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 one row in each of the other tables in T_E. The assignment of values in the tables is similar to Case 1 with the following modifications: (i) p and q are assigned fixed values of 1 and 2, (ii) Each columns $t'.B$ having a path to $t.A$ in JG_E, is assigned fixed values (1, 2) and all other columns of the corresponding table t' are assigned values just like for $t'.X$ in Case 1, except that the assignment is now duplicated across the two rows.

An example D_{gen} for checking the presence of l_orderkey in G_E is shown in Figure 4.2. 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 our EQC restriction to key-based equi-



c_cust key	...
1	

customer

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

lineitem

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

orders

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	

SPJ intermediate result

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

R

Figure 4.2: D_{gen} for Grouping on l_orderkey (Q3)

joins, the above data generation procedure results in ensuring the desired conditions for the intermediate SPJ result. Namely, that it will contain 3 rows with all columns having the same value across these rows 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 turns out to be empty. In this case, we create a D_{gen} with each table having *two rows*, each column in JG_E assigned fixed values $(1, 2)$, and any two different values to all other columns while satisfying all filter predicates. Then, $\mathcal{F}$ is run on this D_{gen}, and if the result contains just one row, we can conclude that the query has an *ungrouped* aggregation.

Overall, the above procedure produces for Q3:

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

4.2 Aggregation Functions

We explain here the procedure for identifying aggregations (*min(), max(), count(), sum(), avg()*) – due to space limitations, we restrict our attention to numeric attributes. However, similar methods can be used for textual/date attributes as well. Further, for ease of presentation, we assume that there is no DISTINCT aggregation – such specialized cases are handled at last.

As described in 3.5, the Projections Extractor extracts each output column as a function of the database columns in its dependency list. For each columns O in $\widetilde{P_E}$, the aggregation identification goes as follows: Let $O = agg(f_o(A_1, ..., A_n))$ where agg corresponds to the aggregation and f_o corresponds to the function identified in Section 3.5. Now our goal is to generate a database D_{gen} such that the *final* result cardinality is **1**, and each of the five possible aggregation functions on f_o results in a *unique value*, thereby allowing for correct identification of the specific aggregation. We call this the "target result".

Since we want to be able to distinguish between *min()* and *max()*, we need at least two different values in the input database columns. Further, to ensure unique values for the various aggregations in the final output, we do the following: Consider a pair of input arguments $(a_1, .., a_i, .., a_n)$ and $(a_1, .., a'_i, .., a_n)$ such that $f_o(a_1, .., a_i, .., a_n) = o_1$ and $f_o(a_1, .., a'_i, .., a_n) = o_2$, with $o_1 \neq 0$, $o_1 \neq o_2$. Note that the two arguments differ only in a_i and a'_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 part 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 constraint:

$$k \notin \left\{ 0, o_1 - 1, o_2 - 1, \frac{o_1 - o_2}{o_1}, \frac{1 - o_2}{o_1 - 1}, \frac{(o_2 - 2) \pm \sqrt{(o_1 - 2)^2 - 4(1 - o_2)}}{2} \right\} \tag{4.1}$$

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. Now, additionally if we ensure that the G_E attributes are assigned common values in all the rows, the result of $\mathcal{F}$ will be the target result.

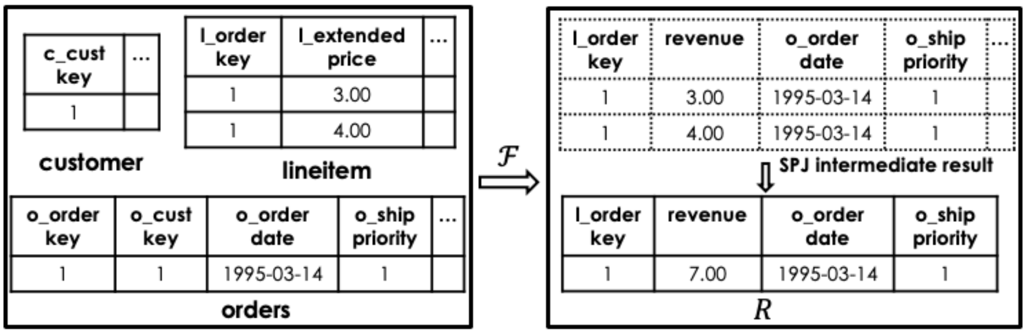
The reason that the target result is produced is (i) the result cardinality is 1 since there is a common set of values for the G_E attributes, and (ii) the constraints on k ensure unique aggregated output of all the aggregations for O. (As a special case, if f_o is a constant function or a function of 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 will be equivalent (e.g. *min(), max(), avg()*), any can be taken as the final choice.

Generating D_{gen}

Firstly, we choose the i^{th} argument A_i to be a column that is not in G_E. If choosing such A_i is not possible, then as mentioned above, $a_i = a'_i$ and any argument column can be chosen as A_i. After that, the data generation process to obtain the above intermediate result for output column $O = agg(f_o(A_1, ..., A_n))$ is similar to the D_{gen} generation of GROUP BY (explained in section 4.1), with the following changes:

- $k+1$ rows are generated for table t where $A_i \in t$, with $t.A_i$ assigned value a_i in k rows and value a'_i in the remaining row.

- With respect to Case 2 ($t.A_i \in JG_E$) in section 4.1, the assignments of fixed values $1, 2$ are replaced with values a_i, a'_i.



c_cust key	...
1	

customer

l_order key	l_extended price	...
1	3.00	
1	4.00	

lineitem

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

orders

l_order key	revenue	o_order date	o_ship priority	...
1	3.00	1995-03-14	1	
1	4.00	1995-03-14	1	

SPJ intermediate result

l_order key	revenue	o_order date	o_ship priority
1	7.00	1995-03-14	1

R

Figure 4.3: D_{gen} for Aggregation on revenue UDF (Q3)

We can either use any two of the arguments that were used to identify dependency list for f_o in Section 3.5 as $(a_1, .., a_i, .., a_n)$ and $(a_1, .., a'_i, .., a_n)$ since they are known assignments that satisfy the required conditions, or generate a new set of arguments. Further, the least positive integer satisfying Equation 4.1 is chosen as k. A sample D_{gen} to check for aggregation on l_extendedprice * (1 - l_discount) is shown in Figure 4.3. Here we set (l_extendedprice, l_discount) as $<(3, 0), (4, 0)>$ and $k = 1$ is feasible.

After getting D_{gen}, we run $\mathfrak{F}$ and the aggregation is identified by matching the result column value (corresponding to O) with the corresponding unique values for the five aggregations. The identified aggregation along with the mapping to the corresponding result column is added to A_E.

At last, entries corresponding to all the aggregated columns are removed from $\widetilde{P_E}$ and inserted in A_E. Further, if there remains an unmapped output column in $\widetilde{P_E}$, it is removed and $count(*)$ is added to A_E. Whatever remains in $\widetilde{P_E}$ now 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}**

Extension to *DISTINCT* keyword

In case the aggregation can be present with *DISTINCT* keyword as well, the following cases may happen as a result of identifying aggregation (without *distinct*) using above method:

Case1 - min() or max() aggregation is identified: In such a case, no action is required as *min()* or *max()* produces exactly same result with/without unique.

Case2 - No aggregation is identified: In such a case, the aggregation on f_o is one of *sum(DISTINCT f_o), avg(DISTINCT f_o)* or *count(DISTINCT f_o).* To identify the correct aggregation, we generate the D_{gen} such that f_o having values (o_1, o_2) such that $o_1 \neq o_2$ and $(o_1 + o_2) \notin \{2, 4\}$ to get value for all three aggregated results unique.

Case3 - Aggregation other than min() or max() is identified: In such a case, the possible actual 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 out this list one by one. For example, let us say that *sum (f_o)* is the identified projection. To prune out 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 out as well. Note that in case of equivalent aggregations, anyone can be chosen.

Extension to non-Numeric Columns

In case of non-numeric column A, we need to find existence of min() or max() only. In such a case, we take k = 1 and take two different values a and b from the domain of A such that the corresponding output column function returns two different values. The rest of the procedure remains same.

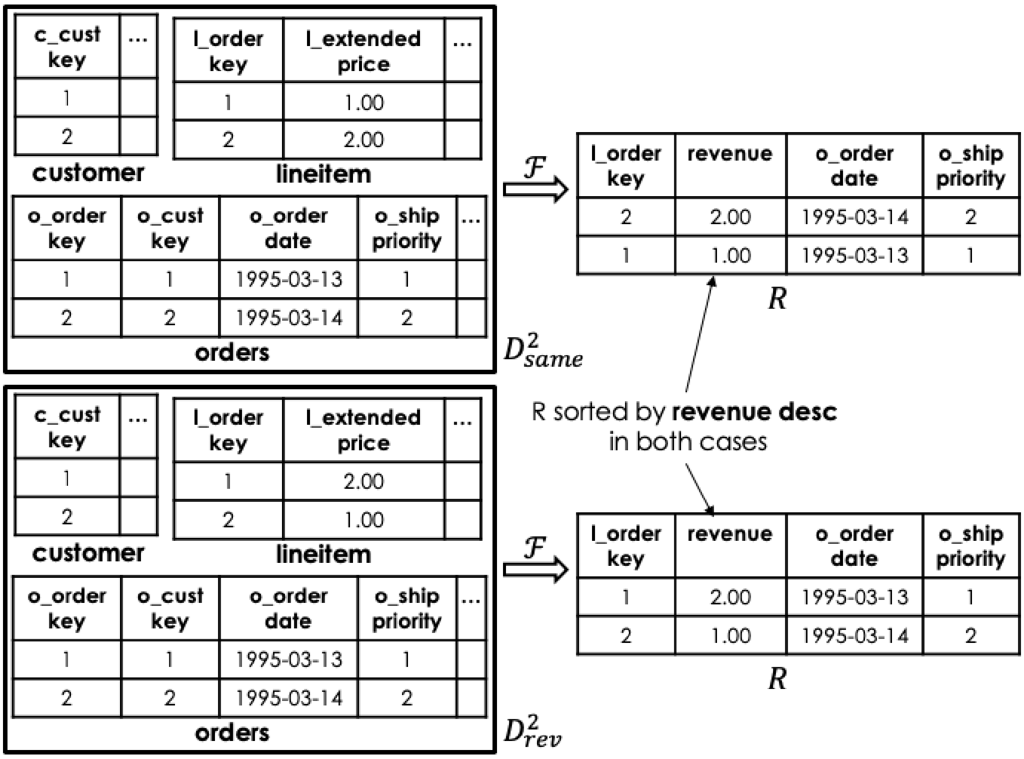
4.3 Order By

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.

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 have been included, or (ii) all functionally-independent attributes



c_cust key	...
1	
2	

customer

l_order key	l_extended price	...
1	1.00	
2	2.00	

lineitem

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

orders

D^2_{same}

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

R

c_cust key	...
1	
2	

customer

l_order key	l_extended price	...
1	2.00	
2	1.00	

lineitem

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

orders

D^2_{rev}

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

R

Figure 4.4: D^2_{same} and D^2_{rev} for Ordering on revenue (Q3)

of G_E have been included in $\overrightarrow{\mathcal{O}_E}$, or (iii) no sort order can be identified for the current index position.

To check for the existence of an output column O, 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 UDF in Figure 4.4.

We use the following procedure to create D^2_{same}: Firstly, we divide the output columns into three sets. S_1, which represents the output columns that are already present in $\overrightarrow{\mathcal{O}_E}$ (initially, $S_1 = \phi$). S_2, which is a singleton set containing the output column that is currently being analyzed. S_3 is the set of all remaining output columns. Let f_o denote the function identified in Section 3.5 for output column O. For each $O \in S_1$, we select a single value for the argument columns which feature in f_o. For each $O \in S_2 \cup S_3$, we select a pair of argument columns such that both the pair return different values for the output column. All these values are generated keeping the filter and join restrictions in consideration. The data generation for all the tables is as follows: (i) Each column that features in S_1 is assigned the single identified value in both the rows. (ii) Each column that features in S_2 and S_3 is assigned the pair of identified values in the two rows so that each output column is sorted in the same order. (iv) For all other columns, two values r and s are assigned such that $r < s$ and both r and s satisfy the associated filter predicate (if any). The key attributes which are connected, get same r and s values. Further, in case of equality filter predicate, we take $r = s$.

The procedure for creating D^2_{rev} is the same as above except that the attributes corresponding to the output column in S_2 are assigned values in the reverse order to that in D^2_{same}.

Database construction in the above manner ensures both the rows form individual groups, so aggregated columns can be effectively treated as projections (except for *count*(), which requires a different mechanism, explained at last). After generating D^2_{same} and D^2_{rev}, we run $\mathcal{F}$ for both the instances and analyze the results. If the values in O are sorted in the same order for both the results, O along with its associated order, is added to $\overrightarrow{\mathcal{O}_E}$ at position i, and the sets S_1, S_2 and S_3 are recalculated for the next iteration.

Lemma 4.1: 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. Now in the result corresponding to D^2_{same}, the values in O will also be sorted in 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. □

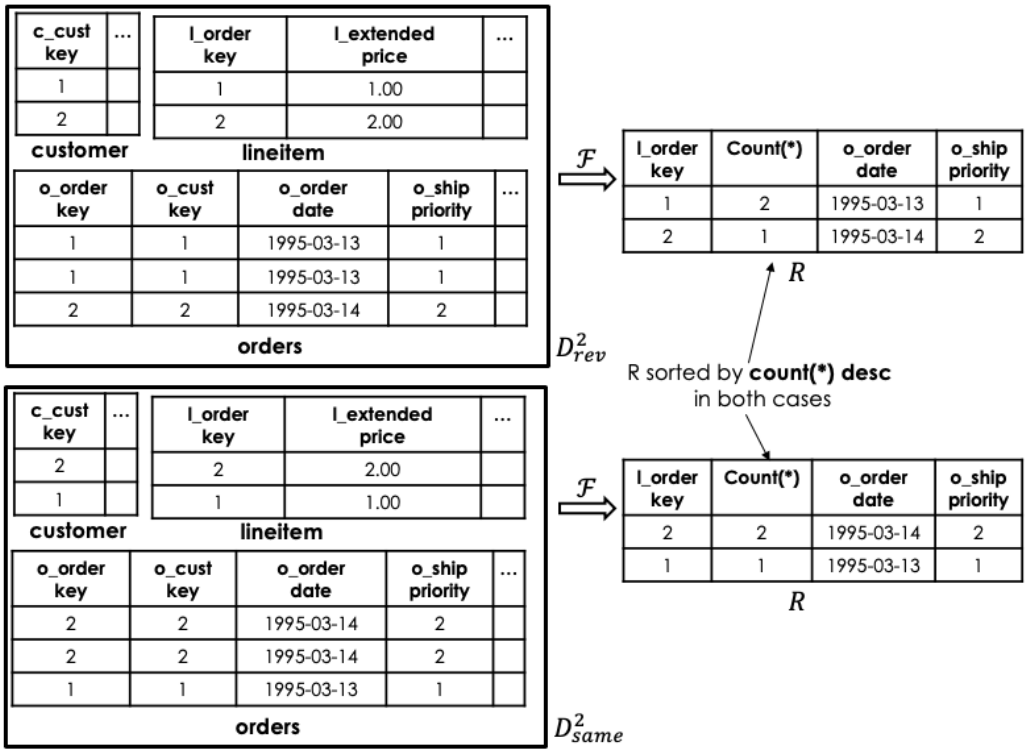
With the above procedure, we finally obtain for Q3:

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

Extension1: *count(*)* $\in A_E$

In the case when *count(*)* $\in A_E$, the two rows in each of the tables is not enough as the count() value for both the groups will be one. In such case, we need an intermediate result (on which grouping will be applied) with 3 rows such that two rows form one group and the third row forms another group. 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 *group by*



c_cust key	...
1	
2	

l_order key	l_extended price	...
1	1.00	
2	2.00	

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

l_order key	Count(*)	o_order date	o_ship priority
1	2	1995-03-13	1
2	1	1995-03-14	2

c_cust key	...
2	
1	

l_order key	l_extended price	...
2	2.00	
1	1.00	

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

l_order key	Count(*)	o_order date	o_ship priority
2	2	1995-03-14	2
1	1	1995-03-13	1

Figure 4.5: D^2_{same} and D^2_{rev} for Ordering on count(*) (Hypothetical scenario:Q3)

clause 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 $p = 1andq = 2$. In other cases, take p and q satisfying the corresponding filter predicates (if any). Note that in the above procedure, if we encounter an attribute with equality filter predicate, we take $p = q = val$ where val satisfies the corresponding filter predicate.

Data generation for D^2_{same} is similar as for D^2_{rev} with the only change being the values of p and q are now swapped. The further procedure of running $\mathcal{F}$ and analyzing the results is the same as explained in *order extraction* part of the section. 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 4.5.

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

Extension2: $t.A : (``t.A = val" \in F_E \wedge (agg\_func(t.A) \in A_E)$

In case there is *min(), max()* or *avg()* aggregation on A, the attribute can be treated as 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 same as in Extension 1. Also note that, the aggregation case with *DISTINCT* keyword is equivalent to non-aggregated projection.

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. In our experiments, we explicitly verified that this was indeed the case.

4.4 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 $\mathcal{F}$ 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.

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 $\mathcal{F}$ 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 $l_E = \mathbf{10}$ for Q3.

Chapter 5

Experimental Evaluation

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 well-provisioned[1] PostgreSQL 11 [20] database platform.

5.1 Hidden SQL Queries

Our first set of experiments was conducted on a representative suite of hidden SPJGAOL queries based on different template queries of the TPC-H benchmark, with the primary change being the removal of nesting; and are similar in complexity to the Q3 running example. For convenience, we hereafter refer to them as Qx, where x is their associated TPC-H query identifier. The exact queries are listed in Appendix A. 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.

we have also run UNMASQUE on (i) the TPC-DS [26] benchmark with PostgreSQL, and (ii) the TPC-H benchmark with SQL Shield encrypted queries on Microsoft SQL Server [19]. The performance results were of a similar nature.

5.1.1 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 component

[1] Intel Xeon 2.3 GHz CPU, 32GB RAM, 3TB Disk, Ubuntu Linux

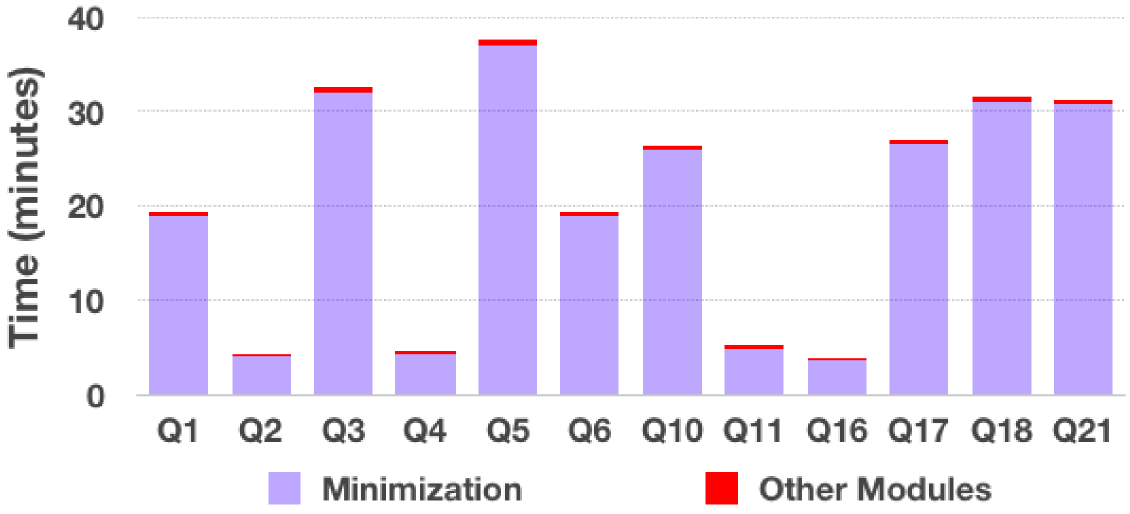


Figure 5.1: Hidden Query Extraction Time (TPC-H 100 GB)

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

5.1.2 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 5.1. 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 these two modules operate on the original large database, whereas, as described in Chapters 3 and 4, the remaining modules work on miniscule mutations or synthetic constructions that contain just a handful of rows. Interestingly, although the executable $\mathcal{F}$ 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.

5.1.3 Optimization

We now go on to show how MINIMIZATION – could be substantially improved with regard to its efficiency.

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 the initial size. Specifically, 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 **Sample Size = 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 optimal tuning of the sample size parameter.

The revised total execution times after incorporating the above two optimizations, are shown in Figure 5.2, 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 5.1. 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 6.7 minutes, shown by the red dashed line in Figure 5.2, 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 [15] or Condenser [24], 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

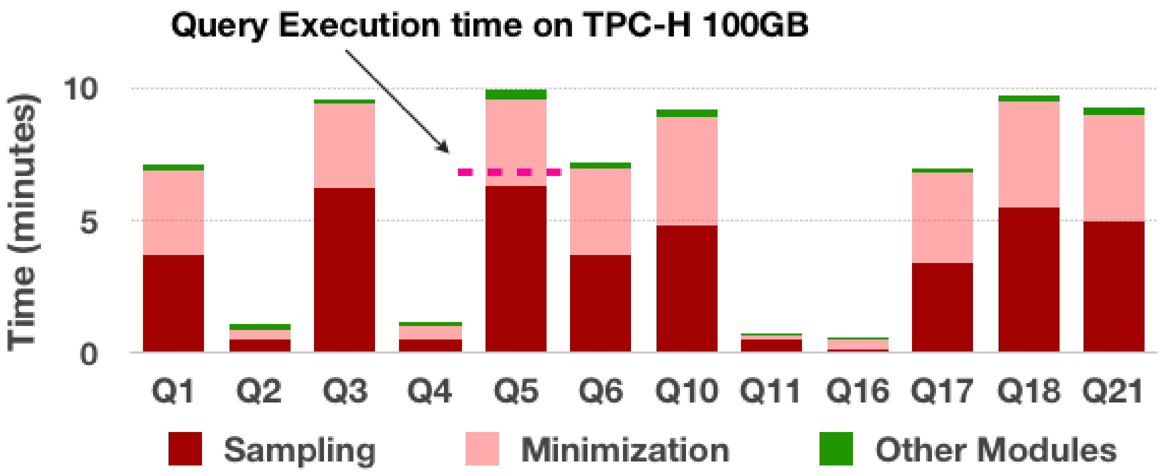


Figure 5.2: Optimized Hidden Query Extraction Time (TPC-H 100 GB)

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.

5.1.4 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 5.3. We see here that all extractions were completed in less than 25 minutes each, demonstrating that the growth of overheads is sub-linear in the database size. In fact, a single query execution of $Q5$ on this database took around 72 minutes, almost 3 times the query extraction time.

5.1.5 TPC-DS Results for 100 GB

The bar-chart in Figure 5.4 shows the time taken to extract 7 queries sourced from TPC-DS benchmark (along with there identifier numbers) on a 100 GB database version. The exact queries are listed in Appendix A. We can see that all the queries were extracted within 4 minutes. It may surprise at first that the time taken in this case is lesser than the time for TPC-H queries and also, the variation amongst queries is very less. The reason is that the table sizes in TPC-DS are not that skewed as in TPC-H. So, no table in TPC-DS is as huge as lineitem table of TPC-H.

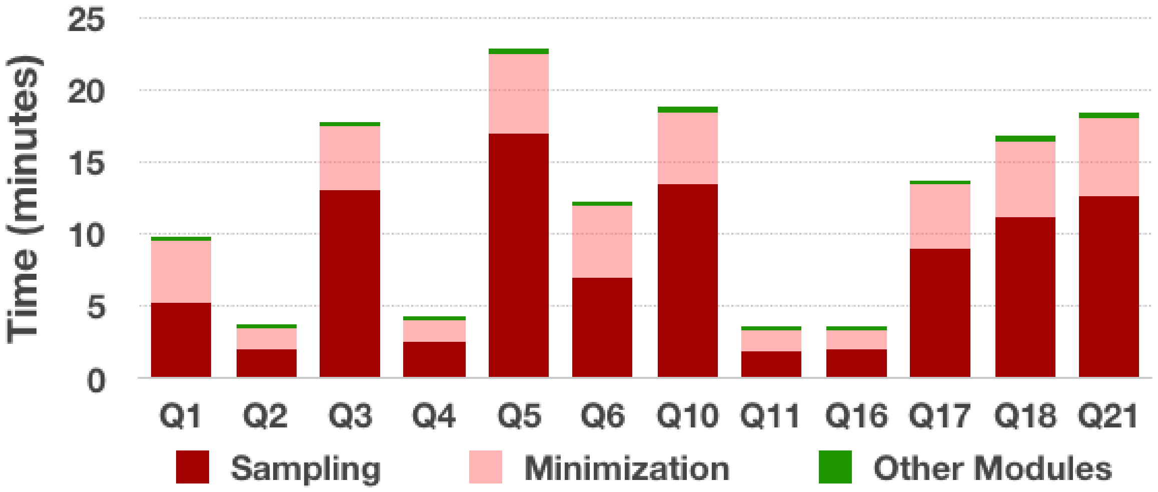


Figure 5.3: Optimized Hidden Query Extraction Time (TPC-H 1 TB)

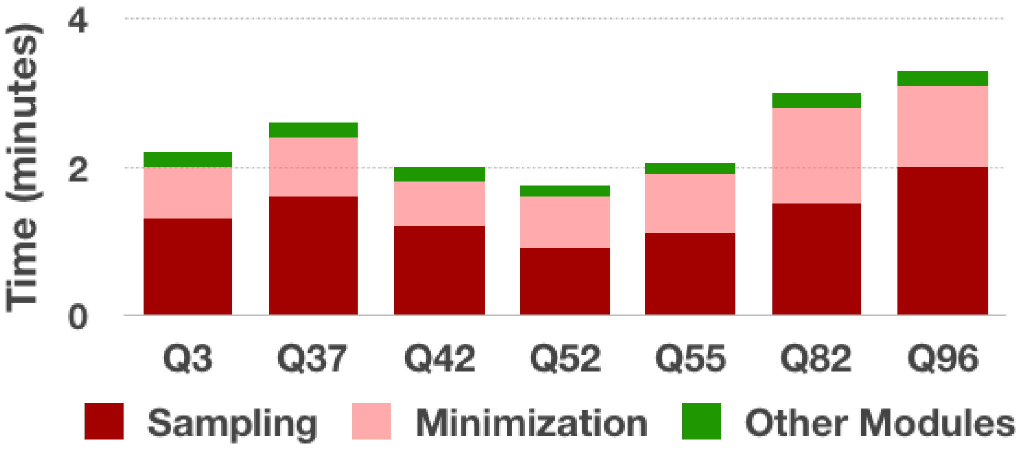
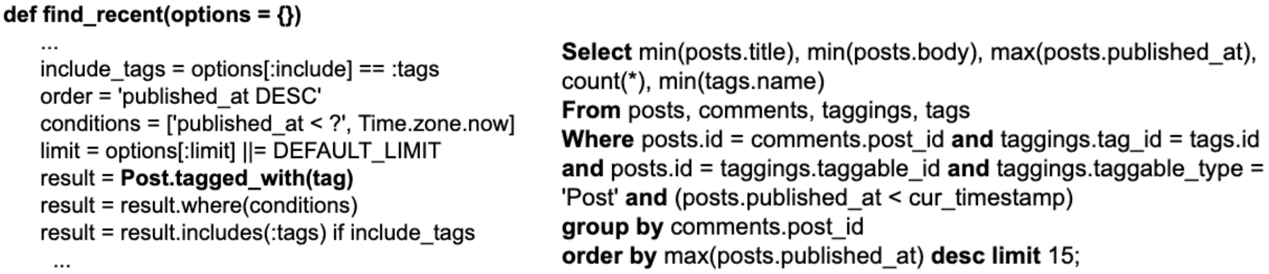


Figure 5.4: Hidden Query Extraction Time (TPC-DS 100 GB)

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

Table 5.1: Imperative to SQL Translation



```
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 5.5: Imperative to SQL Translation

5.2 Hidden Imperative Code

Our second set of experiments evaluated applications hosting imperative code. Here we considered the popular *Enki* [16] and Blog [21] blogging application, both built with Ruby on Rails, each of which has a variety of commands that enable bloggers to navigate pages, posts and comments. 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 [18] 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 5.1 shows the SQL queries extracted w.r.t. the commands. We have omitted five commands 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 5.1 were extracted correctly. As a sample instance, consider the *"get latest posts by tag"* command, a snippet of which is outlined in Figure 5.5a. The corresponding UNMASQUE output is shown in Figure 5.5b, and was produced in just 2.5 seconds.

Chapter 6

Extensions

6.1 Extension to non-integral Key attributes

There are various applications (e.g. Wilos [27]) which use non-integral keys as identifier in the database tables. We assume that the domain of each key attribute contains at least two different values. To handle non-integral keys, the following changes are required:

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 3.3), 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.

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.

6.2 Queries with Having Clause

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 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[1] However,

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

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 Chapter 2, the SPJGA queries with having clause should satisfy the following conditions.

1. The attributes involved in filter predicate in the Having clause and outside Having 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 corresponding data type.

Note: Here, the operation on only integral attributes are discussed. However, the queries with textual attributes (and LIKE operator) can also be handled in a similar manner as defined in previous chapters.

6.2.1 From Clause Detection

From Clause detection is performed in the same way as described in section 3.1.

6.2.2 Database Sampling

If the initial database instance is huge, the sampling (as defined in Chapter 5) is applied to reduce its size. Note that, the whole database is not copied, but a new table is created with the sampled rows. Also "*not null*" constraints are not added in the new table.

6.2.3 Join Graph Detection

Join graph detection is performed in the same way as described in section 3.3. Knowledge of join graph helps reducing the database instance more efficiently. As we can not go with the binary partition argument here, using key relations helps in faster database reduction.

6.2.4 Database Minimization

Given a database instance D and an executable $\mathcal{F}$ producing a populated result on D, derive a reduced database instance D_{min} from D such that removing any row of any table in T_E results in an empty result. We call such database, a minimal database for the query.

With this definition of D_{min}, we can prove the following observations:

Lemma 6.1: *For the EQC, the output of SPJ part of the query for the minimal D_{min} constitutes a **single group** (as per grouping attributes of the query) and the final output contains only a **single row**.* ■

The minimization is done in the following manner: Let t be a table in the From clause (the set T_E) of the query. 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 A. In each iteration, 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 applied to each table in the set T_E repeatedly until the database can not be reduced further. The idea behind the step of preserving a particular value of the attribute is as following: if A is a group by attribute, it will contain a single value in the reduced database instance. Further, we first select the value with the maximum frequency as a heuristic because it selects a relatively large number of rows at a time.

Note that if the query belongs to EQC^{-H}, the final database will 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 not a one row database. We discuss the other case in Section 6.2.10.

6.2.5 Group By Attributes

It is clear by D_{min} construction that any attribute with two or more different values can not be a part of group by clause as it would have created two different groups in the output. So, in order to get the attributes involved in the group by clause, we check for each attribute in the D_{min} which have a single value in all the rows. For each such attribute A with value val_1, we insert each of the current row in the table again with A value being val_2 where $val_2 \neq val_1$. However, val_2 may not satisfy the unknown filter on A, If any. For this, we do this two times, one with $val_2 = val_1 + 1$ and one with $val_2 = val_1 - 1$. Two output rows in any of the two cases indicate A to be present in the group by clause. A similar argument can be used for textual attributes as well.

6.2.6 Having Clause and Filters

First we identify possible filter on each group by column using a similar technique as per in section 3.4. After that, the filters on non-grouping attributes are identified.

For a SPJGA query, filter predicate $a \leq A \leq b$ can be re-written in terms of a having clause condition as $a \leq min(A)$ and $max(A) \leq b$. The procedure below identifies filter predicate in terms of having clause conditions. Thus hereonwards, a filter on A refers to a filter in the form of $val_1 \leq agg\_func(A) \leq val_2$. To detect the having clause condition on an attribute, we change its values in the table, such that only one row of the output group is affected at a time. However, if a foreign key of the table maps to a key of another table in the join graph and values in the foreign key attribute 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 tables. This can be done by traversing the join graph and duplicating rows in the table with new key identifiers. For example, let $T_1[(1, \text{“}a\text{”}, 2), (2, \text{“}b\text{”}, 2)]$ be a table with two rows and $T_2[(2, \text{“}c\text{”})]$ be a table with a single row where last attribute of T_1 refers to the first attribute of T_2. Then these tales are transformed as $T_1[(1, \text{“}a\text{”}, 1), (2, \text{“}b\text{”}, 2)]$ and $T_2[(1, \text{“}c\text{”}), (2, \text{“}c\text{””})]$. Note that both the joins (before and after transformation) produce same output except the key attribute contents.

Let $[i_1, i_2]$ be the integer range. 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. For a filter predicate $val_1 \leq A \leq val_2$, Let us call $A \geq val_1$ as the left filter on A and $A \leq val_2$ as the right filter on A. We first define the term *rowno* and *val*. Starting from 1 to n, if we keep decreasing the value of a_i to i_1, *rowno* denotes the first row, in which the values in A can not be decreased to i_1 without losing the output. Also, $rowno = none$ if values in all the rows can be decreased to i_1. Further, if $rowno \neq none$, *val* denotes the minimum value in row *rowno* which can be present without losing the output. The following algorithm is used to get *rowno* and *val*.

Now, the following two cases arise:

Case 1: $rowno = none$**.** In this case, there is no left filter 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: $rowno \neq none$**.** If $rowno \neq 1$ and $rowno \neq n$, there is a having clause predicate

Algorithm 3: Getting *rowno* and *val* for left filter

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

on A with either $sum() \geq val_1$ or $avg() \geq val_1$. The reason is that, if there were a condition $min(A) \geq val_1$, the value of *rowno* should have been 1. Similarly, if here were a condition $max(A) \geq val_1$, the value of *rowno* should have been n. Now, if $rowno = 1$, the aggregations in the filter predicate may be $sum()$, $avg()$ or $min()$. To differentiate amongsts these, we decrease the value in the first row by 1 and increase the value in any 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 $sum() \geq val_1$ or $avg() \geq val_1$ otherwise it is $min(A) \geq val_1$. A similar method can be used to differentiate amongst $sum()$, $avg()$ or $max()$ when $rowno = n$. The corresponding filter value val_1 will be the *val* obtained from the algorithm.

To find the right filter on A, a similar approach can be used with a new definition of *rowno* and *val*. Starting from n to 1, if we keep increasing the value of a_i to i_2, *rowno* denotes the first row, in which the values in A can not be increased to i_2 without losing the output. Also, $rowno = none$ if values in all the rows can be increased to i_2. Further, if $rowno \neq none$, *val* denotes the maximum value in row *rowno* which can be present without losing the output. The following algorithm is applied to get the *rowno* and *val*.

After getting *rowno* and *val*, right filter can be found in a similar way using the following two cases:

Case 1: $rowno = none$**.** In this case, there is no right filter condition on A.

Case 2: $rowno \neq none$**.** If $rowno \neq 1$ and $rowno \neq n$, there is a having clause predicate on A with either $sum() \leq val_2$ or $avg() \leq val_2$. Now, if $rowno = n$, the aggregations in the

Algorithm 4: Getting *rowno* and *val* for right filter

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

filter predicate may be $sum()$, $avg()$ or $max()$. To differentiate amongsts 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 $sum() \leq val_2$ or $avg() \leq val_2$ otherwise it is $max(A) \leq val_2$. A similar method can be used to differentiate amongsts $sum()$, $avg()$ or $min()$ when $rowno = 1$. The corresponding filter value val_2 will be the val obtained from the algorithm.

Note that we have not yet differentiated between the filters on $sum()$ and filters on $avg()$. Here we make use of the leverage to have *null* values in our database. 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 $\leq$) group by attributes get the same value, 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.

6.2.7 Having condition with count()

After identifying all other filters, the filter with count() can be done in a manner analogous to finding limit in section 4.4.

6.2.8 Projection Clause

The projections are identified in a manner analogous to the method defined in Section 3.5. However, while calculating the function, all the rows of the columns in dependency list are assigned same value and final coefficients are divided by number of rows produced after the join and filters.

6.2.9 Other Clauses

If there is no filter with $count(*)$ in the having clause, we can create a single row database satisfying all the filters. Hence, procedures similar to the ones described for queries in EQC^{-H} can be used. In case of presence of a filter of the form "$count(*)$ *op* k", we add an additional constraint of number of rows for each of the other modules.

6.2.10 One Row database for SPJGHA[OL] query

While database minimization, we may get a one row database for a SPJGA query with Having clause as well. However, to detect Having clause properly, we need 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 6.2.5. After that, in each table, we insert the existing row again with a different key value. If we get a two row output, we can 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. Consider an attribute A containing a value 6 in the database currently. There is a Having clause condition on A 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 output to be non-empty, we place *null* value in a subset of attributes starting from size 1 subsets until we get a non-empty output.

6.2.11 UDF's in Projection

In the absence on a Having clause filter of type $val_1 \leq sum() \leq val_2$, the techniques defined in section 3.5 can be used to detect UDF in the having clause by placing a single unique value in every row of each column. However, in the presence of such filter, 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 UDF.

6.3 Discussion on Other Operators

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.

Chapter 7

Theoretical Results

In this chapter, we prove that for arbitrary queries, Hidden Query Extraction is an *undecidable* problem. We use the following problem to prove the undecidability of HQE.

Semantic Equivalence of queries(SE): Given two arbitrary queries Q_1 and Q_2, determine if Q_1 and Q_2 are semantically equivalent.

Semantic equvalence of two arbitrary SQL queries is a well known undecidable problem [1]. Further, we say that $SE(Q_1, Q_2)$ = true if Q_1 and Q_2 are semantically equivalent, and false otherwise. Before moving on to the main theorem of this chapter, we first state and prove the following lemma.

Lemma 7.1: Let Q_1, Q_2 be two arbitrary queries. For any query Q,

$$(SE(Q_1, Q_2) = true) \implies (SE((Q - Q_1) \cup (Q_1 - Q), (Q - Q_2) \cup (Q_2 - Q)) = true)$$

Proof: **If Part:** Let $SE(Q_1, Q_2) = true$, then for any database instance D, $Q_1(D) = Q_2(D) = R$. Further let $Q(D) = R'$. Then,

$$((Q - Q_1) \cup (Q_1 - Q))(D) = (Q(D) - Q_1(D)) \cup (Q_1(D) - Q(D)) = (R' - R) \cup (R - R')$$

and

$$((Q-Q_2)\cup(Q_2-Q))(D)=(Q(D)-Q_2(D))\cup(Q_2(D)-Q(D))=(R'-R)\cup(R-R')$$

Hence, $SE((Q-Q_1)\cup(Q_1-Q),(Q-Q_2)\cup(Q_2-Q))=true$.

Only If Part: Let $SE(Q_1,Q_2)=false$. It means that there exists a database instance D such that $Q_1(D)\neq Q_2(D)$. WLOG, Let t be a tuple that is present in $Q_1(D)$ but not in $Q_2(D)$. Now there are two possible cases:

(Case 1) $t\in Q(D)$**:** In this case

$$(t\in Q(D)\wedge t\in Q_1(D)\implies t\notin((Q(D)-Q_1(D))\cup(Q_1(D)-Q(D)))$$

and

$$(t\in Q(D)\wedge t\notin Q_2(D),\implies t\in((Q(D)-Q_2(D))\cup(Q_2(D)-Q(D)))$$

Hence, $SE((Q-Q_1)\cup(Q_1-Q),\,(Q-Q_2)\cup(Q_2-Q))=false$.

(Case 2) $t\notin Q(D)$**:** In this case

$$(t\notin Q(D)\wedge t\in Q_1(D))\implies t\in((Q(D)-Q_1(D))\cup(Q_1(D)-Q(D)))$$

and

$$(t\notin Q(D)\wedge t\notin Q_2(D))\implies t\notin((Q(D)-Q_2(D))\cup(Q_2(D)-Q(D)))$$

Hence, $SE((Q-Q_1)\cup(Q_1-Q),\,(Q-Q_2)\cup(Q_2-Q))=false$. □

Now, we prove that the Hidden Query extraction problem is undecidable in general.

Theorem: For an arbitrary hidden query (denoted as black-box function F), Hidden Query Extraction (HQE) problem is Undecidable.

Proof: Suppose that HQE is decidable. Then, there exists a deterministic algorithm A such that for any database instance D and a function F with $F(D) \neq \phi$, $A(F, D)$ produces a Q_{out} which is semantically equivalent to unknown hidden query in F. Further, let us say $F_1 = H(Q_1)$ and $F_2 = H(Q_2)$ where H is some function (or a wrapper) which simply hides the query. To continue our proof, we state and prove the following two lemmas first.

Lemma 7.2: Q_1 and Q_2 are semantically equivalent iff $F_1 = F_2$.

Proof: F_1 and F_2 can be seen as relations which relate the set of database instances to a set of result instances. If F_1 and F_2 represent the exact same relation, Q_1 and Q_2 will be semantically equivalent otherwise there would exist at least one D, which is mapped to different results in F_1 and F_2. Similarly, the other direction can be proved. □

Lemma 7.3: For any F_1, F_2 and D, $(F_1 = F_2) \implies (A(F_1, D) = A(F_2, D))$. Also, $(A(F_1, D) = A(F_2, D)) \implies (F_1 = F_2)$.

Proof: The first statement holds because A is deterministic algorithm. The second statement can be proved by contradiction. Let us say that $A(F_1, D) = A(F_2, D)) = Q$ but $F1 \neq F_2$. Now, Q will be semantically equivalent to the query in F_1 and F_2 both while $F_1 \neq F_2$, a contradiction for Lemma 7.1. □

Let Q_1, Q_2 be two arbitrary queries. **We prove that if HQE is decidable, there exists a deterministic algorithm for** $SE(Q_1, Q_2)$ **for arbitrary queries** Q_1 **and** Q_2. WLOG, let us say that Q_1 and Q_2 are compatible in *set difference* ('$-$') and *set union* ('$\cup$') operations. If this is not the case, we can say that Q_1 and Q_2 are not equivalent. Further, let Q be any simple project join query which is compatible with Q_1 and Q_2 for '$-$' and '$\cup$' operations. Note that such query Q can be generated easily after observing the result column data types of Q_1 or Q_2. Also, let D be a database instance such that $Q(D) \neq \phi$. Note that, although finding such D is a hard problem in general, it can be easily done in case of a project join query. We define two black-box functions as follows:

$$F_1 = H((Q - Q_1) \cup (Q_1 - Q))$$

and

$$F_2 = H((Q - Q_2) \cup (Q_2 - Q))$$

WLOG, let us say that D gives same non-empty result on F_1 and F_2. Note that if this is not the case, then by Lemma 7.1, Q_1 and Q_2 are not semantically equivalent. Further, if D gives empty result in both cases, it means that D gives a non-empty result on Q_1 and Q_2. In such a case, we take $F_1 = Q_1$ and $F_2 = Q_2$ and keep the D as it is. Now that we have F_1, F_2 and D, using Lemma 7.2 and Lemma 7.3, we can say that

$$(A(F_1, D) = A(F_2, D)) \implies (F_1 = F_2) \implies SE(Q_1, Q_2)$$

Thus, presence of a deterministic algorithm for HQE shows the presence of a deterministic algorithm for SE. As SE is known to be undecidable, so is HQE.

□

Chapter 8

Conclusion and Future Work

We introduced and investigated the problem of Hidden Query Extraction as a novel version of QRE, which has a variety of real-world use-cases. As the first step toward solving this problem, we presented the UNMASQUE algorithm, which is based on a combination of database mutation and database generation pipelines. An attractive feature of UNMASQUE is that it is completely non-invasive, facilitating its deployment in a platform-independent manner.

UNMASQUE is capable of identifying a large class of hidden SPJGHAOL queries, similar to those present in the decision-support benchmarks. 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 miniscule 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 current work, we are attempting to extend the scope of EQC to include the Having clause in a more general sense. Also, a mathematical analysis to help choose the appropriate SZ setting for sampling. For the long-term, the extraction of nested queries and outer joins poses a formidable challenge. More fundamentally, characterizing the extractive power of non-invasive techniques is an open theoretical problem.

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995. 45

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

[3] A. Cheung, A. Solar-Lezama and S. Madden. Optimizing Database-Backed Applications with Query Synthesis. In *Proc. of ACM PLDI Conf.*, 2013. 1

[4] 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 1

[5] D. Kalashnikov, L. Lakshmanan and D. Srivastava. FastQRE: Fast Query Reverse Engineering. In *Proc. of ACM SIGMOD Conf.*, 2018. 1

[6] G. Karvounarakis, Z. Ives and V. Tannen. Querying Data Provenance. In *Proc. of ACM SIGMOD Conf.*, 2010. 4, 10

[7] K. Panev and S. Michel. Reverse Engineering Top-k Database Queries with PALEO. In *Proc. of EDBT Conf.*, 2016. 1

[8] 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

[9] W. Tan, M. Zhang, H. Elmeleegy and D. Srivastava. Reverse Engineering Aggregation Queries. *PVLDB*, 10(11), 2017. 1

[10] W. Tan, M. Zhang, H. Elmeleegy and D. Srivastava. REGAL+: Reverse Engineering SPJA Queries. *PVLDB*, 11(12), 2018. 1, 3

[11] Q. Tran, C. Chan and S. Parthasarathy. Query Reverse Engineering. *The VLDB Journal*, 23(5), 2014. 1, 2

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

[13] C. Wang, A.Cheung, R. Bodik. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proc. of ACM PLDI Conf.*, 2017. 1

[14] M. Zhang, H. Elmeleegy, C. Procopiuc, and D. Srivastava. Reverse Engineering Complex Join Queries. In *Proc. of ACM SIGMOD Conf.*, 2013. 1

[15] Jailer: www.jailer.sourceforge.net/home.htm 33

[16] https://github.com/xaviershay/enki 36

[17] docs.microsoft.com/en-us/sysinternals/downloads/strings 6

[18] https://www.selenium.dev/ 36

[19] www.microsoft.com/en-in/sql-server 31

[20] www.postgresql.org 5, 31

[21] https://rubyonrails.org/ 36

[22] www.softwareheritage.org/mission/software-is-fragile 2

[23] www.sql-shield.com 6

[24] Condensor: www.tonic.ai/post/condenser-a-database-subsetting-tool/ 33

[25] www.tpc.org/tpch/ 3

[26] www.tpc.org/tpcds/ 31

[27] Wilos: an orchestration process software. www.openhub.net/p/6390 6, 37

Appendix A

A.1 Experiment Queries 1 (Based on corresponding TPC-H 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 $\leq$ 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

Selecto_orderdate, o_orderpriority, count(*) **as** order_count

From orders

Where o_orderdate $\geq$ date ‘1997-07-01’ **and** o_orderdate $<$ date ‘1997-07-01’ $+$ interval ‘3’ month

Group By l_orderkey, 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 $\geq$ 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 $\geq$ 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 $\geq$ 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;
```

A.2 Experiment Queries 2 (Based on corresponding TPC-DS 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 ≥ 30 and
household_demographics.hd_dep_count = 3 and store.s_store_name = 'ese'
Order by count(*)
Limit 100;
```