Efficient Extraction of Hidden Negation Predicates

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

BY

Mukul Sharma

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

June, 2022

Declaration of Originality

I, **Mukul Sharma**, with SR No. **04-04-00-10-42-20-1-17935** hereby declare that the material presented in the thesis titled

Efficient Extraction of Hidden Negation Predicates

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2020-22**.
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: June, 2022 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 Family and Friends

for their love and support

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. I am thankful to him for his valuable guidance and moral support. I feel lucky to be able to work under his supervision.

I am thankful to Anupam Sanghi for valuable inputs and constructive criticism. I am also thankful to Kapil Khurana for mentoring and assisting me on numerous occasions. I would also like to thank all my lab mates for their constant support. I am grateful to Aman Sachan and Abhinav Jaiswal for helping me with the project and for making my lab time fun.

Finally, I would like to thank my parents and brother, who have always supported me indefinitely.

Abstract

Database queries present in applications can be hidden due to explicit encryption or complex internal representation. Unmasking the hidden queries within the database applications is termed as *Hidden Query Extraction*(HQE) problem in [1]. The diverse use-cases of this problem range from resurrecting legacy code to query rewriting. UNMASQUE algorithm is a first step towards addressing the HQE problem. It is a non-invasive, platform-independent extraction algorithm that extracts SQL queries hidden within database applications.

There are two main issues within UNMASQUE, from the performance and extraction point of view. They inefficiently handle the input-output database tables. We incorporated techniques for the efficient handling of input-output database tables. We are extending the scope of UNMASQUE by including Not Equal Predicates into its extractable domain. A detailed evaluation on synthetic benchmarks demonstrates the minimization in extraction overhead and accurate extraction of *Not Equal* (<>) hidden queries.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Database queries embedded within applications may become invisible due to explicit or implicit opacity. Encryption or obfuscation may have been incorporated to protect the application logic. Alternatively, opacity may also arise when the application source code is written in a complex manner like ORM translations. These queries are termed as *Hidden Queries* in [1] and the functions as *executable* for the queries. Hidden-Query Extraction (HQE) was recently introduced in [1], and its task is to identify the hidden query.

Formally defined, HQE is: *Given a black-box application $\mathcal{A}$ containing a Hidden query $\mathcal{Q}_\mathcal{H}$(in either SQL format or its imperative equivalent), and a database instance $\mathcal{D}_\mathcal{I}$ on which $\mathcal{A}$ produces a populated result $\mathcal{R}_\mathcal{I}$, unmask $\mathcal{Q}_\mathcal{H}$ to reveal the original query (in SQL format).* The goal of Hidden Query Extraction is to find the precise $\mathcal{Q}_\mathcal{H}$ such that $\forall_i \mathcal{Q}_\mathcal{H}(D_i) = R_i$.

A ground-truth query is additionally available but in a hidden form that is not easily accessible. HQE has a variety of use cases such as: Imperative Code to SQL Translation, Debugging Application with stored SQL procedures, Recovering of lost source code etc. UNMASQUE (Unified Non-invasive MAchine for Sql QUery Extraction) is a first step towards addressing the HQE problem. It is a platform-independent hidden query extractor introduced in [1]. It extracts the hidden query $\mathcal{Q}_\mathcal{H}$ through "active learning" - that is, by using the outputs of application executions on carefully crafted database instances. It uses a judicious combination of database mutation and database generation techniques to extract hidden queries. UNMASQUE is capable of extracting a basal set of warehouse queries that feature the core SPJGAOL (Select, Project, Join, Group by, Aggregation, Order by, Limit) clauses.

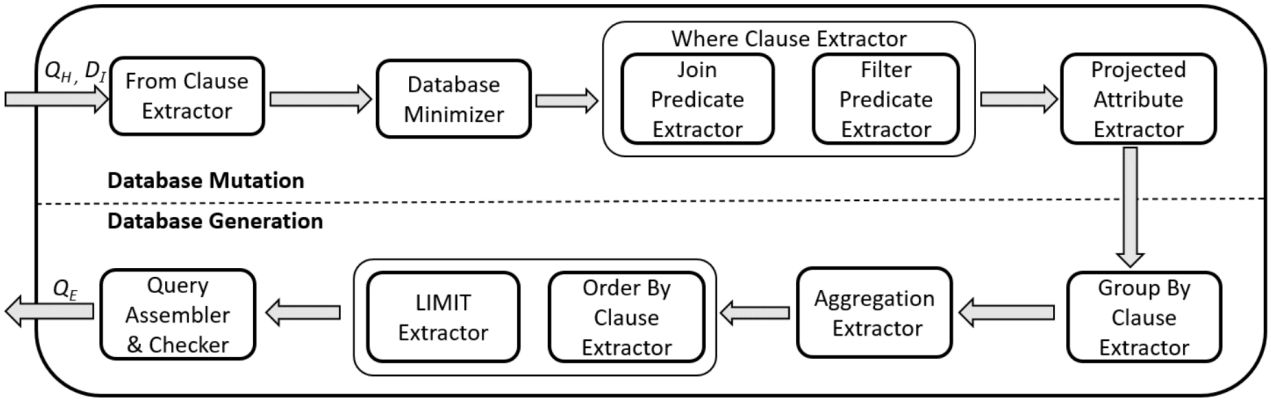

Figure 1.1: UNMASQUE Architecture

1.1 Motivation

Our motivation for incorporating Not Equal predicate into hidden query extraction as it is a commonly used operator in database applications. It is evident from the widely used business benchmarks. Among the 22 queries in the TPC-H benchmark [3], as many as 5 performs Not Equal operation. Among the 99 queries in the TPC-DS benchmark [4], 11 contains Not Equal operators. We will be using the following query running example throughout to explain subsequent report sections. UNMASQUE will not be able to extract the following hidden query $\mathcal{Q}_\mathcal{H}$ because it contains *Not Equal* (<>) and *Not Like* operator, and hence it falls outside of the class of extractable queries. We will use *NEP* as the abbreviation to represent the Not Equal predicate.

Select *l_shipmode, count(*) as count*
From *lineitem*
Where *l_quantity* > 20 *and l_quantity* <> 25
and l_shipmode ***not like*** '%AIR%'
Group By *l_shipmode*
Order By *l_shipmode*

The MINIMIZER module of the UNMASQUE pipeline, shown in Figure 1.1, takes up the lion's share(more than 98 %) of the extraction time. The extreme skewness across the modules because the minimizer operates on the original large database, whereas the other modules work with minuscule databases. We can make the entire extraction fast by making the MINIMIZER

module efficient.

1.2 Technical Challenges

The primary challenge in Not Equal predicate extraction is identifying the single value from the wide search space of attribute. It is not feasible to search *NEP* value in the huge domain range of each attribute. We have devised a minimization technique that will quickly reduce the search space of the attribute to find the *NEP* value.

The other challenges in addressing the HQE problem are due to the inherent complexities of the query clauses and the acute dependencies between them. The extraction of *NEP* is challenging due to its dependence on Filter Extraction Module of Figure 1.1. The filter extractor may extract a subsumed filter predicate if we include column <> val in their extractable domain. We have discussed this challenge in detail in Section 2.2.

1.3 Our Contribution

We extend the current scope of HQE by including Not equal into the extractable domain and enhancing its efficiency. The key design principles are:

***NEP* Extraction**

This project takes under consideration of UNMASQUE *Extractable Query Class*(EQC) assumptions and tries to expand its domain by including the ablity to handle <> and *Not Like* operator. Our contribution is to unmask the hidden queries containing <> and *Not Like* operators under certain assumptions.

Views-based Minimizer

One of the key modules of UNMASQUE is the Data Minimizer module. It identifies the smallest subset D^1 of $\mathcal{D}_I$ such that $\mathcal{Q}_\mathcal{H}$ continues to produce the populated result on this D^1. This minimization module is the bottleneck in the current pipeline because it operates on the original large database $\mathcal{D}_I$, whereas the other modules work with a miniscule database D^1. The minimization module times increase as the size of the original database instance increases. We have proposed a View-based Minimizer, which will take no extra disk space and improve the minimization time by orders of magnitude.

Result Comparator

The result comparator of the query checker module in UNMASQUE compares the output results of the hidden query $\mathcal{Q}_\mathcal{H}$ and extracted query $\mathcal{Q}_\mathcal{E}$ to verify the correctness of extracted query. The UNMASQUE comparison algorithm follows a brute force approach in which each tuple

of one table is linearly searched into another table. It will slow down when the cardinality of the tables is high. There is also a correctness issue with the above approach. The algorithm will produce an incorrect answer when the unique tuples are identical, but the frequency of the duplicate tuples in the two tables is different. We have proposed some faster algorithms for results comparison. They will compute the hash of the tables and then compare their hash values for table equivalence check.

1.4 Performance Evaluation

We have evaluated *NEP* extraction behavior on complex SQL queries containing <> and *Not Like*, arising in a synthetic TPC-H environment. These complex queries are derived from the popular TPC-H benchmark. Our experiments, conducted on a Google Cloud platform, indicate that the hidden queries are precisely identified in a timely manner. As a case in point, the running example query is extracted on a 100 GB TPC-H instance within 10 minutes.

1.5 Organization

The remainder of the paper is organized as follows: The background work is reviewed in Section 2. The problem framework is discussed in Section 3. Further, the solution overview for *NEP* extraction is introduced in Section 4, and then described in detail in Sections 5 through 8. The experimental evaluation is reported in Section 9. The related work is discussed in Section 10. Finally, our conclusions and future research avenues are summarized in Section 11. The appendix contains some proofs and experimental test queries.

Chapter 2

BACKGROUND

We have discussed some of the prerequisites of UNMASQUE, which will help to understand the *NEP* extraction. The class of queries that UNMASQUE can handle is defined in [1] as *Extractable Query Class* (*EQC*). *EQC* assumes : (i) $\{<>, Not\ Like\}$ are out of the Extractable Query Class. (ii) Filter predicate features only non-key columns and are of the type *column op value*. For numeric columns $op \in \{=, \leq, \geq, <, >, between\}$ and for textual columns $op \in \{=, like\}$; (iii) The join graph is a sub-graph of the schema graph(comprised of all valid PK-FK and FK-FK edges); (iv) All the ordering columns appear in the projections; (v) The limit value is at least 3; (vi) All joins are key based equi-joins.

2.1 Database Minimization

Database Minimizer identifies the smallest subset D_{min} of the initial database instance $\mathcal{D}_I$ such that hidden query $\mathcal{Q}_\mathcal{H}$ continues to produce the populated result on this D_{min}. For *EQC* class hidden queries, there will always exist single-row minimized tables. These single-row D_{min} are referred to as D^1 in [1]. It uses a recursive database partitioning technique to find D^1. It picks a table t from T_E(Set of tables in $\mathcal{Q}_\mathcal{E}$) that contains more than one row and divides it roughly into two halves. Run $\mathcal{Q}_\mathcal{H}$ on the first half, and if the result is populated, retain only this partition. Otherwise, retain only the second half. Eventually, all the tables of T_E have been reduced to a single row by this process.

2.2 Filter Extraction

As per *EQC*, the filter predicates should be on non-key attributes and are of the type *column op value*. The following steps are taken to check the presence of the filter predicate on attribute A. Let $[i_{min}, i_{max}]$ be the value spread of column A in the integer domain, and the range predicate

in $\mathcal{Q}_{\mathcal{H}}$ is of the form $l \leq A \leq r$, where l and r need to be identified. All the comparison operators $(=, <, >, \leq, \geq, between)$ can be represented in this format (e.g. $A < 5 \equiv i_{min} \leq A \leq 4$).

Table 2.1: Filter Predicate Cases

Case	$\lvert R_1 \rvert = \phi$	$\lvert R_2 \rvert = \phi$	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
1	No	Yes	$i_{min} \leq A \leq r$	Find r
1	Yes	Yes	$l \leq A \leq r$	Find l and r

Replace the value of column A with i_{min} in $\mathcal{D}^1$ and then execute $\mathcal{Q}_{\mathcal{H}}$. Similarly, replace the value of column A with i_{max} in $\mathcal{D}^1$ and then execute $\mathcal{Q}_{\mathcal{H}}$. If we get a non-empty output result, in either case, the filter predicate is present in column A. If the match is with Case 2, a binary-search is conducted over $(i_{min},\ a]$ to identify the value of l, where a is the value of column A present in D^1. Similarly, for Case 3, the search is over $[a,\ i_{max})$ to identify the value of r. Finally, Case 4 is a trivial combination of Cases 2 and 3, and handled in a similar manner.

2.3 Result Comparator

In the final module, UNMASQUE conducts a suite of automated tests to verify the extraction correctness. First, several randomized large databases are created on which both the application and the extracted query are run. The results are compared, and a non-zero difference indicates an error. We denote R_H and R_E as the result of the hidden query and extracted query, respectively. The UNMASQUE comparison algorithm follows a brute force approach in which each element of result R_E is linearly searched into R_H. This approach will slow down when the cardinality of the output results is high.

Chapter 3

Problem Framework

In this section, we summarize the basic problem statement, and the underlying assumptions of the *NEP* extraction.
Problem Statement: The hidden query $\mathcal{Q}_{\mathcal{H}}$ contains Not Equal predicates, unmask $\mathcal{Q}_{\mathcal{H}}$ to reveal the original query such that $\forall i$, $\mathcal{Q}_{\mathcal{H}}(\mathcal{D}_I) = \mathcal{R}_I$.

3.1 Assumptions

We define a new class of supported queries called Extractable Query Class with $NEP(EQC^N)$. In addition to *EQC* assumptions, we require some additional mild assumptions to achieve this extended coverage of Not Equal Predicate.

- The Not Equal Predicate present in the filter is only on the non-key columns and is of the type *column op value* where $op \in \{<>, not\ like\}$.
- When we execute hidden query $\mathcal{Q}_{\mathcal{H}}$ on the database instance $\mathcal{D}_{\mathcal{I}}$, each *NEP* present in $\mathcal{Q}_{\mathcal{H}}$ must separately impact the output result.
- There can be at most one *NEP* present per attribute. Extraction of the *NOT IN* operator is out of our extractable domain.
- The possible wildcard characters that are used with the Not Like operator are '_' and '%.'

The second assumption is crucial to identify *NEP* efficiently. If *NEP* doesn't show any visible impacts on the original database $\mathcal{D}_I$, we cannot identify *NEP* efficiently.

For ease of presentation, we are assuming one *NEP* per attribute. The modifications made in the filter extractor of UNMASQUE (Section 5) can handle one *NEP* per attribute. We can

extend the same idea to k *NEP* per attribute, i.e., *NOT IN* operator.

3.2 Notations

The main acronyms and key notations used in the rest of the paper are summarized in Table 3.1.

Table 3.1: Notations

Symbol	Meaning(wrt query Q_E)
$\mathcal{E}$	Application Executable
$\mathcal{D}_I$	Initial Database Instance
$\mathcal{D}_{min}$	Reduced Database
$\mathcal{D}_{mut}$	Mutated Database
$\mathcal{Q}_{\mathcal{H}}$	Hidden Query
$\mathcal{Q}_{\mathcal{E}}$	Extracted Query
$\mathcal{R}_H$	Result of Hidden Query
$\mathcal{R}_E$	Result of Extracted Query
T_E	Set of tables in Q_E
F_E	Set of Filter predicates in Q_E

Chapter 4

Solution Overview

In this section, we overview the core design principles of *NEP* extraction, with the running example of the Introduction used as a query to explain each module overview. Subsequently, in Sections 5 through 8, all the architecture modules are described in detail.

The extracted query $\mathcal{Q}_\mathcal{E}$ from UNMASQUE is given as input to our *NEP* extraction module. When our running example query is given as input to the current UNMASQUE(with the modifications of Section 5), all the query components will get successfully extracted except <> and *Not Like* operators. The extracted query $\mathcal{Q}_\mathcal{E}$ will be as follows:

Select *l_shipmode*, *count(*) as count*
From *lineitem*
Where *l_quantity* $>$ 20
Group By *l_shipmode*
Order By *l_shipmode*;

The Figure 4.1 shown above is the architecture for *NEP* extraction. All of UNMASQUE's

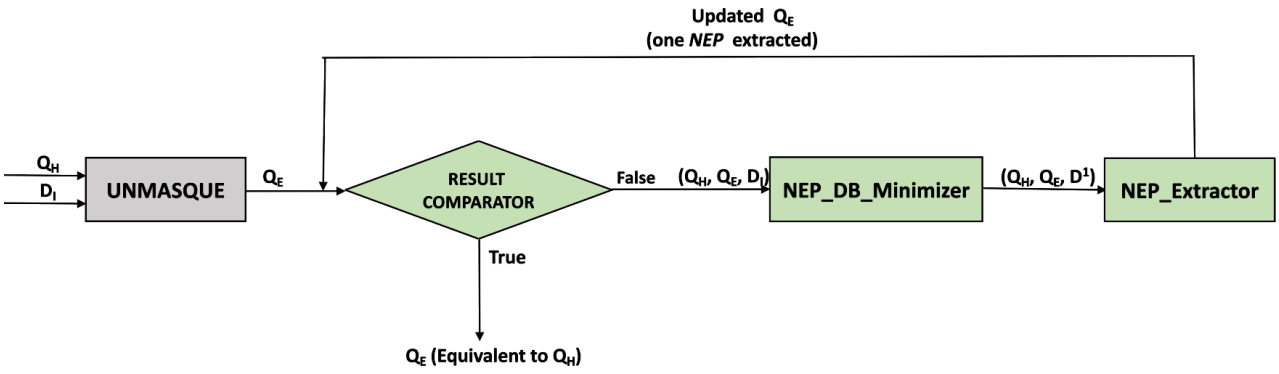

Figure 4.1: NEP Architecture

original code-base, other than the changes in *Filter Extraction* module, was used as a black-box. The modification in the filter extractor clause is discussed in Section 5. We will use the UNMASQUE pipeline extracted output query $\mathcal{Q}_\mathcal{E}$ for *NEP* extraction. The key observation helpful to extract *NEP* is the difference in the output results from the hidden query $\mathcal{Q}_\mathcal{H}$ and extracted query $\mathcal{Q}_\mathcal{E}$. From these output results, we can detect whether the *NEP* is present and find attribute values on which it is present.

The Result Comparator module will detect whether the Not Equal Predicate is present in $\mathcal{Q}_\mathcal{H}$ or not. If the hidden query $\mathcal{Q}_\mathcal{H} \in EQC^N$ and extracted query $\mathcal{Q}_\mathcal{E}$ produces different results on $\mathcal{D}_I$, *NEP* is present in $\mathcal{Q}_\mathcal{H}$.

Result Comparator module will work as *NEP* detector. The hidden query $\mathcal{Q}_\mathcal{H}$, extracted query $\mathcal{Q}_\mathcal{E}$ and initial database $\mathcal{D}_I$ are given as the input to the *NEP_DB_Minimizer* module. This module will find a reduced database $\mathcal{D}^1$ from $\mathcal{D}_I$ such that $\mathcal{Q}_\mathcal{E}$ gives a populated result and $\mathcal{Q}_\mathcal{H}$ gives an empty result on $\mathcal{D}^1$. The reduced database $\mathcal{D}^1$ for our running example is shown in Table 4.1.

Table 4.1: Minimized lineitem table I

l_orderkey	*l_shipdate*	*l_commitdate*	*l_quantity*	*l_shipmode*
10	1993-09-10	1994-02-05	25	MAIL

The reduced database instance $\mathcal{D}^1$ is given as an input to the *NEP_Extractor* module. The *NEP_Extractor* module will extract one *NEP* at a time using database mutation techniques. This module will extract *l_quantity* <> 25 for our running example and updates the extracted query by including extracted *NEP*. The updated extracted query $\mathcal{Q}_\mathcal{E}{}'$ will be:

Select *l_shipmode*, *count(*) as count*
From *lineitem*
Where *l_quantity* > 20 and *l_quantity* <> 25
Group By *l_shipmode*
Order By *l_shipmode*;

Using this updated extracted query $\mathcal{Q}_\mathcal{E}{}'$ again, the presence of some other *NEP* is checked. This cycle will repeat until all the *NEP* gets extracted from $\mathcal{Q}_\mathcal{H}$. The *not like* operator will get extracted for our running example query in the next cycle. In the following sections, we present the internal details of each of the aforementioned concepts.

Chapter 5

Modified Filter Extractor

If *column* <> *val* is present in the hidden query $\mathcal{Q}_{\mathcal{H}}$, then the filter extractor may extract a subsumed filter predicate. We can understand this problem from our running example. Suppose we are extracting the filter extractor on the *l_quantity* attribute. The range of column *l_quantity* attribute is [1, 50], and the value of column *l_quantity* in D^1 is 50. As we have discussed the working of filter extractor in Section 2.2. A binary search is conducted over [1, 50] to find the value of l. Due to the presence of *l_quantity* <> 25 in $\mathcal{Q}_{\mathcal{H}}$, a subsumed filter predicate $A \geq 26$ will get extracted. Due to the presence of *NEP*, the binary search finds the incorrect value of l.

As per our EQC^N assumptions defined in Problem Framework Section, there can be at most one *NEP* per attribute in $\mathcal{Q}_{\mathcal{H}}$. By putting one additional check at **Line 8** of **Algorithm 1**, we can avoid finding subsumed filter predicate. When we find the value of l, at any iteration of Binary Search, if we get an empty result for the *middle* value and a populated result for the $middle - 1$ value, then it means that *NEP* is present on the *middle* value.

If the condition of **Line 9** in the following algorithm is true, then *NEP* is present on the *mid* value. In our running example, $\mathcal{Q}_{\mathcal{H}}$ produces empty result on D^1 when the value of *l_quantity* attribute is 25 and non-empty results when the value of *l_quantity* attribute is 24. It means that *l_quantity* <> 25 is present in $\mathcal{Q}_{\mathcal{H}}$. Similarly, when we find the value of r, we have put an additional check for one larger value of *mid*. This modification ensures the correct working of the UNMASQUE Filter extractor module under EQC^N assumptions.

The following algorithm is the modified filter extraction algorithm of UNMASQUE under EQC^N assumptions:

Algorithm 1: Modified Filter Extraction

Data: $T_E, \mathcal{D}^1$
Result: F_E
$left \leftarrow i_{min}$
$right \leftarrow \mathcal{D}^1.A$
while $left < right$ **do**
  $mid \leftarrow (left + right)/2$
  $\mathcal{D}^1.A \leftarrow mid$
  $r_1 \leftarrow \mathcal{Q}_{\mathcal{H}}(D^1_{mut})$
  $\mathcal{D}^1.A \leftarrow mid - 1$
  $r_2 \leftarrow \mathcal{Q}_{\mathcal{H}}(D^1_{mut})$; `// Additional check`
  if $r_1 = \phi$ *and* $r_2 \neq \phi$ **then**
    $F_E.add(`A <> mid')$
    $right \leftarrow mid$
  end
  else if $r_1 = \phi$ *and* $r_2 = \phi$ **then**
    $left \leftarrow mid + 1$
  end
  else
    $right \leftarrow mid$
  end
end
$l \leftarrow left$
$F_E.add(`A \geq l')$
return F_E

Chapter 6

NEP Database Minimizer

To extract every Not Equal Predicate present in $\mathcal{Q}_{\mathcal{H}}$, We invoke the *NEP_DB_Minimizer* module to minimize the database as far as possible while maintaining different output results on extracted query$\mathcal{Q}_{\mathcal{E}}$ and hidden query $\mathcal{Q}_{\mathcal{H}}$. If we can find such minimized single-row database, we can extract *NEP* by database mutation techniques discussed in the *NEP_Extractor* module.

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

Lemma 6.1.1 *For the EQC^N, there always exists a $\mathcal{D}^1$ identified by* NEP_DB_Minimizer *on which $\mathcal{Q}_{\mathcal{H}}$ and $\mathcal{Q}_{\mathcal{E}}$ produce different output results.*

Proof of Correctness: Given an initial database instance $\mathcal{D}_I$, extracted query $\mathcal{Q}_{\mathcal{E}}$ from UNMASQUE, and hidden query $\mathcal{Q}_{\mathcal{H}}$ containing *NEP*, our *NEP_DB_Minimizer* module will always identify reduced database instance D^1.

Firstly, since the output result from $\mathcal{Q}_{\mathcal{H}}$ and $\mathcal{Q}_{\mathcal{E}}$ are different on $\mathcal{D}_1$, the intermediate output results obtained after the evaluation of the *SPJ* core of the query are also guaranteed to be different. 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 on which $\mathcal{Q}_{\mathcal{H}}$ and $\mathcal{Q}_{\mathcal{E}}$ give the different results because: (i) If there is no row from table t, r_i cannot be derived because the inner equi-join with table t will result in an empty output from both $\mathcal{Q}_{\mathcal{H}}$ and $\mathcal{Q}_{\mathcal{E}}$.(ii) If there are k rows ($k > 1$) from table t on which $\mathcal{Q}_{\mathcal{H}}$ and $\mathcal{Q}_{\mathcal{E}}$ give different outputs, $(k-1)$ rows either do not satisfy one or more join/filter F_E predicates and can therefore be removed, or they will produce a result of more than one row. We can trace back to the single-row database(containing *NEP* value) on which $\mathcal{Q}_{\mathcal{H}}$ and $\mathcal{Q}_{\mathcal{E}}$ give different output results.

We will discuss the overview of the following Algorithm 2. It works similar to the database minimization algorithm of UNMASQUE with the changes in **line number 9**. It picks a table T from T_E that contains more than one row and divides it roughly into two halves. Run the Match algorithm on the first half, and if it returns false, retain only this partition. Otherwise, retain only the second half. The Match algorithm will efficiently compare the result of extracted query $\mathcal{Q}_\mathcal{E}$ with the result of hidden query $\mathcal{Q}_\mathcal{H}$. The output of the Match algorithm is true if $\mathcal{R}_\mathcal{H}$ and $\mathcal{R}_\mathcal{E}$ are equivalent otherwise false. Each time after table halving, Match Algorithm is executed.

The following *NEP_DB_Minimizer* algorithm is used to identify reduced database instance $\mathcal{D}^1$:

Algorithm 2: *NEP_DB_Minimizer*

```
   Data: Q_H, Q_E, D_I
   Result: D^1
 1 D^1 ← φ
 2 foreach Table T in T_E do
 3 |   while | T |> 1 do
 4 |   |   Divide T into two halves T_u and T_l
 5 |   |   T ← T_u
 6 |   |   D_J.update(T)
 7 |   |   R_H ← Q_H(D_J)
 8 |   |   R_E ← Q_E(D_J)
 9 |   |   if not Match(R_H, R_E) then
10 |   |   |   drop T_l
11 |   |   end
12 |   |   else
13 |   |   |   drop T_u
14 |   |   |   T ← T_l
15 |   |   end
16 |   end
17 |   D^1.add(T)
18 end
19 return D^1
```

6.2 Correctness

Lemma 6.2.1 *For EQC^N class of assumptions,* NEP_DB_Minimizer *will correctly identify a reduced database $\mathcal{D}^1$ such that $\mathcal{Q}_\mathcal{E}(D^1) = \phi$ and $\mathcal{Q}_\mathcal{H}(D^1) \neq \phi$.*

Proof of Correctness: Under EQC^N assumptions, if $\mathcal{R}_{\mathcal{H}}$ and $\mathcal{R}_{\mathcal{E}}$ are different on $\mathcal{D}_I$, then *NEP* must present in $\mathcal{Q}_{\mathcal{H}}$ as the other UNMASQUE modules don't get impacted due to the presence of *NEP*. *NEP_DB_Minimizer* will find a reduced database $\mathcal{D}^1$ from $\mathcal{D}_I$ such that $\mathcal{R}_{\mathcal{H}} = \mathcal{Q}_{\mathcal{H}}(D^1)$ and $\mathcal{R}_{\mathcal{E}} = \mathcal{Q}_{\mathcal{E}}(D^1)$ are different. As the cardinality of input reduced database $\mathcal{D}^1$ is one, the cardinality of the output results $\mathcal{R}_{\mathcal{H}}$ and $\mathcal{R}_{\mathcal{E}}$ must be at most one. This implies that some attribute of $\mathcal{D}^1$ contains *NEP* value. Therefore, the cardinality of $\mathcal{R}_{\mathcal{H}}$ must be zero because $\mathcal{D}^1$ has the *NEP* value.

Since $\mathcal{R}_{\mathcal{H}}$ and $\mathcal{R}_{\mathcal{E}}$ are different and $\mathcal{R}_{\mathcal{H}}$ is empty, $\mathcal{R}_{\mathcal{E}}$ must be non-empty. Hence, *NEP_DB_Minimizer* will identify a $\mathcal{D}^1$ such that $\mathcal{R}_{\mathcal{H}}$ is empty and $\mathcal{R}_{\mathcal{E}}$ is non-empty. From Lemma 6.1.1 and 6.2.1, we can conclude that the *NEP_DB_Minimizer* algorithm will always identify the reduced database instance $\mathcal{D}^1$ such that $\mathcal{Q}_{\mathcal{E}}(D^1) = \phi$ and $\mathcal{Q}_{\mathcal{H}}(D^1) \neq \phi$. We can identify all *NEPs* from $\mathcal{D}^1$ by simple mutation techniques.

6.3 Optimizations in MINIMIZER

The MINIMIZER module of the UNMASQUE pipeline takes up more than 98 % of the extraction time. It is a bottleneck in the UNMASQUE pipeline. Database minimization is a one-time process in UNMASQUE. But in *NEP* extraction, for each *NEP* extraction database minimization is done. Therefore, to make *NEP* extraction time practical, we need to work on the efficiency of minimizer.

We can see in **Algorithm 2** that in each iteration of the while loop, we are creating a new table by copying the upper half of the current table. This copying of tables will incur time and space overhead. We can avoid this extra time and space overhead by avoiding the materialization of the tables. We have used virtual views for table halving rather than materializing a new table every time.

We have used systems tuple identifiers to create views. A tuple identifier represents a physical location of a row. They are database vendor dependent. For, e.g., *ctid* in PostgreSQL and *rowid* in Oracle. It is the fastest way of locating a row. A *Ctid* of a row is represented as a pair (block number, tuple number within block). The *Ctid* of the first row of the table is '(0, 1)'. The number of tuples present in a block is table-width dependent. Based on the number of tuples per block, we can estimate the *ctid* of the middle row of the table. Suppose one block contains 50 tuples, then the following query will create a view T_u which contains roughly the upper half of table T:

Create View T_u as

Select * ***From*** T ***Where*** *ctid* $\geq$ '(0, 1)' *and ctid* $\leq$ '($|T|$/100, 1)' ;

The view T_u roughly contains the upper half of table T. The approximate block number of the middle row of table T would be $|T|/50$ * 1/2. Using *Ctid's*, we can quickly locate the required chunk of large tables. We can query the view through hidden query $\mathcal{Q}_{\mathcal{H}}$. View creation is a constant-time operation. The extra time and space of materializing the table T_u is avoided.

If the database vendor doesn't provide the system's tuple identifier, we create an extra column in the schema that acts as pseudokey. We will create an index on this column and create the views T_u using this extra column instead of *ctid*. This Approach will work similarly to the previous tuple identifier approach.

We have performed the empirical analysis of minimization time for both the UNMASQUE and Views Approach in the experiments section. These experiments justify the above theoretical claims. We have also recorded the extra disk space for both approaches to verify no extra disk requirement in the Views Approach. The result comparator module internally uses Match Algorithm for table equivalence check. We have described the working of an efficient Match Algorithmin the next section.

6.4 Time Complexity

We assume a simple cost model, defined as follows: Let $|T|$ denote the size of table T, measured in terms of the row-cardinality. Then, the time to run a query that includes m tables (say T_1, T_2, ..., T_m) is directly proportional to the product of the table sizes. *NEP_DB_Minimizer* algorithm will recursively halve the selected table through Views (which is constant-time operation) and then execute the hidden query. Every time, the query cost is reduced by half. The time taken by the *NEP_DB_Minimizer* algorithm to reduce the table to a single row can be computed as $(\frac{\tau}{2} + \frac{\tau}{4} + \ldots + 1)$, which is upper bounded by $O(\tau)$.

Chapter 7

RESULT COMPARATOR

The Match procedure of Algorithm 2 will compare the result $\mathcal{R}_{\mathcal{H}}$ of the hidden query with the result $\mathcal{R}_{\mathcal{E}}$ of the extracted query $\mathcal{Q}_{\mathcal{E}}$. For a query in EQC^N, the matching algorithm will work as a Not Equal Predicate detector. If results $\mathcal{R}_{\mathcal{H}}$ and $\mathcal{R}_{\mathcal{E}}$ are different, then it means that some *NEP* is not extracted. The UNMASQUE comparison algorithm follows a quadratic approach in which each tuple of result $\mathcal{R}_{\mathcal{E}}$ is linearly searched into result $\mathcal{R}_{\mathcal{H}}$.

We have proposed some methods which have lower overheads for result comparison. We have devised two algorithms for the result comparison, i.e., Faster Comparison-based and Computation-based algorithms. Tables $\mathcal{R}_{\mathcal{E}}$ and $\mathcal{R}_{\mathcal{H}}$ are explicitly created from extracted query $\mathcal{Q}_{\mathcal{E}}$ and hidden query $\mathcal{Q}_{\mathcal{H}}$ outputs, respectively.

7.1 Comparison-based Methods

Each tuple's presence in the other table is checked in a comparison-based algorithm. We have used the 'Except All' SQL operator for result comparison. We will count the number of rows left after subtracting the result tables $\mathcal{R}_{\mathcal{E}}$ and $\mathcal{R}_{\mathcal{H}}$ from each other.

$C_1 \leftarrow$ **Select** count(*) **From** (**Select** * **From** $\mathcal{R}_{\mathcal{E}}$
Except All Select * **From** $\mathcal{R}_{\mathcal{H}}$) T;
$C_2 \leftarrow$ **Select** count(*) **From** (**Select** * **From** $\mathcal{R}_{\mathcal{H}}$
Except All Select * **From** $\mathcal{R}_{\mathcal{E}}$) T;

If the value of both C_1 and $C2$ equals zero, $\mathcal{R}_{\mathcal{E}}$ and $\mathcal{R}_{\mathcal{H}}$ are equivalent. The benefit of using the SQL operator for result comparison is that the database optimizer will intelligently switch

the query plan having less cost. When the cardinality of the result is less, it uses the Sort Merge plan; Otherwise, it uses Hashing. We have used 'All' because of the possibility of duplicate rows in $\mathcal{R}_{\mathcal{E}}$ and $\mathcal{R}_{\mathcal{H}}$.

7.2 Computation-based Methods

In this method, a hash is generated row-wise or table-wise, and then these hash values are compared for result comparison. If aggregated hash values of table $\mathcal{R}_{\mathcal{E}}$ and $\mathcal{R}_{\mathcal{H}}$ are equal, they are equivalent. The contents of each tuple of the table are not compared. We have discussed two types of computation-based methods:

1. **Table Hash Method:** We compute the hash value of the entire result tables $\mathcal{R}_{\mathcal{E}}$ and $\mathcal{R}_{\mathcal{H}}$. If the computed hash value is equal, the result tables are equivalent. The drawback of the table hash method is that before computing hash, we need to sort both the tables on all attributes of $\mathcal{R}_{\mathcal{E}}$ and $\mathcal{R}_{\mathcal{H}}$. The sorting will ensure the same tuple ordering in both tables. Firstly, we need to apply the 'Order By' on all the attributes and compute the table hash. PostgreSQL has many Hash functions; we have used the HashText hash function. HashText function calculates a 32-bit signed integer hash value of the input string.

2. **Row Hash Method:** We compute the integer hash values of the individual tuples of the table, then take the aggregate sum of all these integer hash values. We will call this as **checksum** value of a table. If the checksum value of both the table $\mathcal{R}_{\mathcal{E}}$ and $\mathcal{R}_{\mathcal{H}}$ are equal, then it means that the tables are equivalent. The advantage of using the Row hash method is that there is no need to sort the tables. The checksum value with or without sorting would be the same. We have used the same HashText function for calculating the checksum value.

We have conducted the empirical analysis of execution time for all the above result comparison techniques in the experiments section. The Table Hash method is the slowest because of the extra cost of sorting the tables on all attributes. The Row hash method is the fastest result comparison technique. We have used the Row Hash method as our Match algorithm.

We have compared the results comparison time of the UNMASQUE approach with our fastest Row Hash approach. The comparison time in the UNMASQUE approach increases exponentially as the result cardinality increases. We have improved the output results comparison time by orders of magnitude. Match procedure algorithm is frequently executed in Algorithm 2. The row hash method reduces the overall *NEP* comparison time significantly.

The following algorithm is our final Result Comparison Algorithm:

Algorithm 3: Match_Algorithm

Data: $\mathcal{R}_{\mathcal{H}}, \mathcal{R}_{\mathcal{E}}$

Result: Comparison of $\mathcal{R}_{\mathcal{H}}, \mathcal{R}_{\mathcal{E}}$

$H_1 \leftarrow$ Sum of all rows integer hash values of $\mathcal{R}_{\mathcal{H}}$

$H_2 \leftarrow$ Sum of all rows integer hash values of $\mathcal{R}_{\mathcal{E}}$

if $H_1 == H_2$ **then**

  return True ; // $\mathcal{R}_{\mathcal{H}}$ and $\mathcal{R}_{\mathcal{E}}$ equivalent

end

else

  return False ;

end

Chapter 8

NEP Extraction

The reduced database instance $\mathcal{D}^1$ from *NEP_DB_Minimizer* is given as input to the *NEP_Extractor* module. Using database mutation techniques *NEP_Extractor* module will extract the Not Equal Predicate source attribute and its corresponding value. The filter extractor module of UNMASQUE extracts all the filter and join predicates under *EQC* assumptions. Therefore, we can refer to the values of the attribute which satisfy the corresponding filter and join predicates of $\mathcal{Q}_\mathcal{E}$, and are termed as filter-compliant values.

NEP_Extractor algorithm will iteratively explore all the attributes of $\mathcal{D}^1$. It will mutate each attribute value with some other filter predicate compliant value and then run $\mathcal{Q}_\mathcal{H}$. If $\mathcal{Q}_\mathcal{H}$ gives a populated output result, it means that the present attribute is the source *NEP* column, and the associated value present in $\mathcal{D}^1$ before the mutation is the *NEP* value. We are mutating the attribute values of $\mathcal{D}^1$ with the values that satisfy the corresponding filter and join predicates in the $\mathcal{Q}_\mathcal{E}$.

The following procedure for *NEP* exploration is skipped for equality predicate attributes. If the hidden query $\mathcal{Q}_\mathcal{H}$ contains both equality and not equal predicate, i.e., $A = m1$ and $A <> m2$. It is equivalent to $A = m1$. There is no need to find *NEP* for equality predicate attributes. *NEP_Extractor* module updates $\mathcal{Q}_\mathcal{E}$ by including the extracted *NEP*. This cycle of *NEP_DB_Minimizer* and *NEP_Extractor* repeats until all the *NEP* of different attributes gets extracted.

The following *NEP_Extraction* algorithm is used to extract the *NEP* from $\mathcal{D}^1$:

Algorithm 4: NEP_Extractor

Data: $\mathcal{Q}_{\mathcal{H}}, \mathcal{Q}_{\mathcal{E}}, \mathcal{D}^1$
Result: $\mathcal{Q}_{\mathcal{E}}$ containing *NEP*

```
foreach Table T in T_E do
    flag ← 0
    foreach Attribute A in T do
        val1 ← T.A
        T.A ← val2 ;    /* Update the value of A with other filter compliant
         value  */
        if Q_H(D^1_mut) ≠ φ then
            F_E.add('A <> val1')
            Q_E.update(F_E)
            flag ← 1
            break
        end
    end
    if flag == 1 then
        break
    end
end
return Q_E
```

8.1 Textual *Not Like* predicate

The *NEP* Extraction will work correctly for numeric, decimal, boolean, and date data types. 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 '%'). *NEP_DB_Minimizer* will find a string that satisfies the actual filter value. In our running example, the 'AIR' string will be extracted from D^1. We aim to extract the actual filter value from this string if *Not Like* is present in the where clause of $\mathcal{Q}_{\mathcal{H}}$. The reduced database $\mathcal{D}^1$ for *Not Like* extraction in our running example will be:

Table 8.1: Minimized lineitem table II

l_orderkey	*l_shipdate*	*l_commitdate*	*l_quantity*	*l_shipmode*
12	1994-12-10	1996-11-05	22	AIR

8.1.1 Identify MQS

The following algorithm for finding the actual filter value of Not Like operator is similar to Algorithm 2 defined in [2]. The only possible wildcard characters that are used with the *Not Like* operator are '_' and '%.' The basic logic in the algorithm is that when we replace or remove a character in the string of an attribute in $\mathcal{D}^1$ and then run $\mathcal{Q}_\mathcal{H}$. If we get a non-empty result on this mutated ${D^1}_{mut}$, then this character is part of the actual filter value of *Not Like*. We will define Minimal Qualifying String(MQS) – given a character string expression *str*, its MQS is the string obtained by removing all occurrences of '%' from *str*. For example, "AIR_" is the MQS for "%AIR_%." The following algorithm will identify MQS using the string value of Column A in $\mathcal{D}^1$, denoted as *rep_str*.

Algorithm 5: Identifying MQS

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

The idea here is to loop through all the characters of *rep_str* and determine whether it is present as an intrinsic character of MQS or invoked through wildcards ('_' or '%'). Replace each character of *rep_str* in $\mathcal{D}^1$ with a different character and then execute $\mathcal{Q}_\mathcal{H}$ on this mutated

database. If the result is non-empty, the replaced character is part of MQS. Otherwise, that character was invoked through wildcard characters.

After obtaining the MQS, we need to find the location of the '%' wildcard character. We will linearly select each pair of consecutive characters in MQS, and a random character that is different from both these characters is inserted between them. Then, we replace the current value in attribute A with this new string. The non-empty result of $\mathcal{Q}_{\mathcal{H}}$ on this mutated database instance indicates the existence of '%' between the pair of characters. The inserted character is removed after each iteration and we start with the initial MQS for each successive pair of consecutive characters. It is done to ensure that the character length limit for A is not exceeded. In the case of our running example MQS for *l_shipmode* attribute will be 'AIR' and the actual filter value will be '%AIR%'.

Lemma 8.1.1 *For a query in EQC^N, Algorithm 5 will correctly identify MQS for the* Not Like *operator on the textual attribute.*

Proof of Correctness: The correctness of Algorithm 5 can be established using contradiction. For example, let us say a character 'a' belonged to MQS, but the procedure fails to identify it. After removing 'a' from *rep_str*, the result is still empty (the filter condition for *not like* was satisfied). It is only possible when 'a' occurs more than once in *rep_str* and at least one occurrence is part of the replacement for wildcard '%'. However, the procedure will keep removing 'a' until there is no occurrence left which is part of the 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. This proof is similar to the identification of MQS for the *Like* operator in [2].

8.1.2 Time Complexity to identify MQS

If *len* is the character limit of the textual attribute, then the time complexity of the Algorithm 5 is O(*len*). Because the algorithm will linearly iterate the *rep_str* string, and in each iteration, it will perform constant-time operations. After finding MQS, one more single pass is required to find the location of the '%' wildcard character. Therefore, the time complexity for actual filter value extraction of the *Not Like* operator is linear in the maximum number of characters allowed by the textual attribute.

8.2 Time Complexity *NEP* Extraction

We have already discussed the time complexity of *NEP_DB_Minimizer* module in Section 6.4. It is upper bounded by $O(\tau)$. In the *NEP_Extractor* module, we are iterating over all the attributes and performing a constant-time operation. The time complexity for this module will be $O(m)$, where m is the total number of attributes in all the tables.

If k *NEPs* are present in the hidden query $\mathcal{Q}_{\mathcal{H}}$, then this cycle of *NEP_DB_Minimizer* and *NEP_Extractor* repeats k times. Therefore, the overall time complexity is $O(k(\tau + m))$.

Chapter 9

Experiments

We now move on to empirically evaluating Minimizer, Result Comparator, and *NEP* extraction efficiency. Our experiments are carried out on the google cloud platform, installed PostgreSQL 11 database (Intel Xeon 2.3 GHz CPU, 32GB RAM, 3TB Disk, Ubuntu Linux) with default primary-key indices. Our experiments cover the accuracy, time and space overhead aspects of *NEP* extraction.

9.1 Copy-based Minimizer Vs Views-based Minimizer

The proposed optimized approach of database minimization using views is implemented in Python 3.6 and integrated with the UNMASQUE pipeline. We have compared the Minimizer time of UNMASQUE with the Views approach. We also compared the extra disk space requirement of UNMASQUE and Views Approach.

9.1.1 Empirical Analysis of Running Time

We have conducted experiments on different sized TPC-H databases and compared the execution times. The hidden query $\mathcal{Q}_{\mathcal{H}}$ on which the minimization times are reported is the TPC-H $\mathcal{Q}_1$ query. We have used a stacked bar chart to show the table creation time and rest minimizer time separately.

We can see that the blue part (table creation time) shares the maximum chunk of minimization time in UNMASQUE. The views approach is significantly faster because there is no table creation part. On large databases like 100GB, UNMASQUE took close to 4500 seconds, whereas the Views approach completed the minimization in only 320 seconds. The minimization time improved by orders of magnitude.

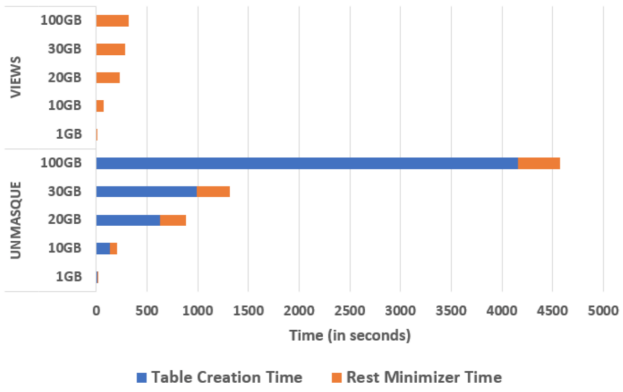

Figure 9.1: UNMASQUE and Views Comparison Time

9.1.2 Empirical Analysis of Disk Space

We have recorded the extra disk space required by both copy-based and views-based minimizer. The total extra disk space required by UNMASQUE is dependent on the size of the initial database instance $\mathcal{D}_I$. The views-based minimizer avoids the extra disk space required for materializing the tables. There is no requirement for disk space in the proposed optimized minimizer. We have used this optimization in *NEP* extraction.

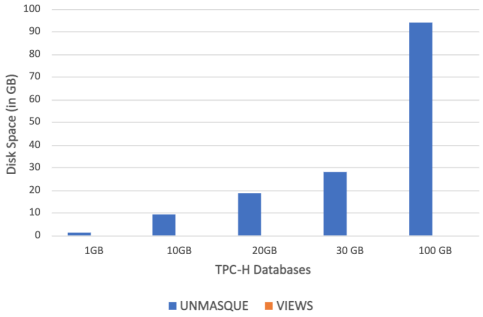

Figure 9.2: UNMASQUE and Views Space Overhead

9.2 Result Comparator Techniques

We have conducted the experiments to compare the three Result comparison techniques i.e., Comparison-based, Table Hash and Row Hash method. The execution time is recorded for each technique with different cardinality of the result tables. There are 16 attributes in each result table.

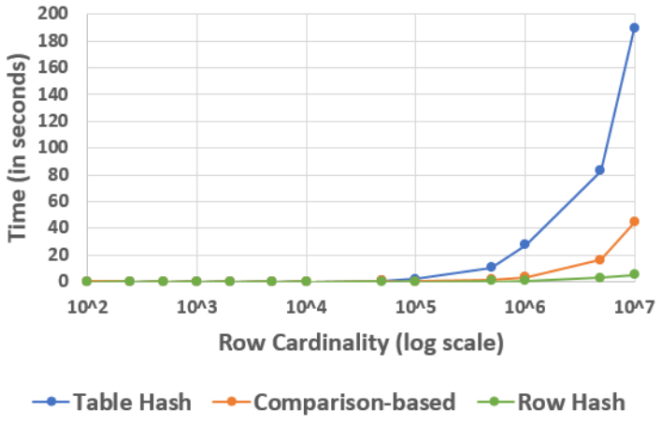

Figure 9.3: Performance Comparison

The Table Hash method is the slowest because of the extra cost of sorting the tables on all attributes. The sorting cost will increase as the number of attributes of the result table increases. The computation-based methods are faster than comparison-based methods because computing a hash is a one-time process, whereas checking each row's presence is expensive. The computation-based Row Hash technique is the fastest result comparator method. We have used it as our final Match algorithm.

We have compared the results comparison time of the UNMASQUE approach with our fastest Row Hash approach.

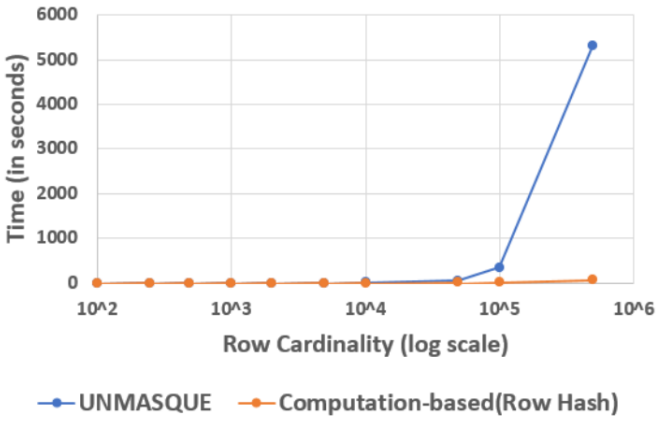

Figure 9.4: UNMASQUE and Row-Hash Comparison Time

We have compared the result comparison time of our faster row hash approach with the UNMASQUE approach. The comparison time in the following graph also contains additional time required to create the output result $\mathcal{R}_{\mathcal{H}}$ and $\mathcal{R}_{\mathcal{E}}$ tables. We have improved the output

results comparison time by orders of magnitude, specifically when the size of result tables is large. The result comparator has a crucial role in *NEP* extraction.

9.3 *NEP* Extraction time

We implemented the proposed *NEP* algorithm in Python 3.6 and integrated with the existing UNMASQUE codebase. The new modules were tested against a set of *NEP* queries to verify the correctness and to see how much overhead is incurred due to the additions. The experiments are conducted on a basal suite of EQC^N class queries. All these complex queries contain *Not Equal* operators. These queries are derived from the popular TPC-H benchmark queries. To conduct a better evaluation, we need complexity in queries that TPC-H provides. We have reported the extra overhead incurred due to the addition of this new module. We have incorporated the View-based minimizer and modified filter extractor in the UNMASQUE codebase. The UNMASQUE's codebase with modifications was used as a black box in *NEP* extraction.

All these derived benchmark queries are listed in the appendix. The total end-to-end time taken to extract each of the 12 queries on a 100 GB initial instance (with a populated result) is shown in Figure 9.5. In addition, the breakup of the *NEP* extractor module and UNMASQUE module execution time is shown in the Figure.

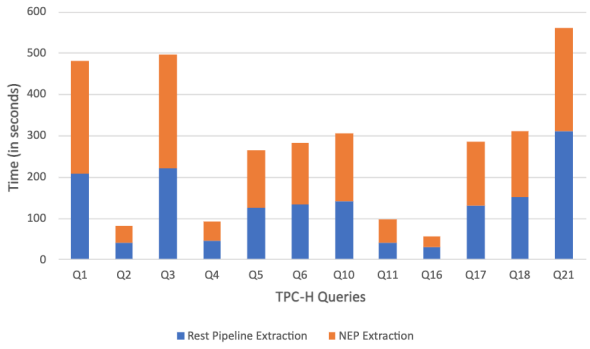

Figure 9.5: Time Breakup for SF 100

We have done the manual verification of all the output extracted queries. The extraction times are practical for offline analysis environments, with all extractions being completed within 10 minutes. When we drilled down into the performance profile, both Minimizer and *NEP_DB_Minimizer* take up the maximum shares of extraction time. The Minimizer module operates on the large database tables, where as the rest modules work with miniscule databases.

Chapter 10

Related Work

Over the past decades, a variety of novel approaches have been proposed for the *query reverse-engineering* (QRE) problem. The general QRE problem statement is: Given a database instance $\mathcal{D}_I$ and a populated result $\mathcal{R}_I$, identify a candidate SQL query $\mathcal{Q}_C$ such that $\mathcal{Q}_C(\mathcal{D}_I) = \mathcal{R}_I$. This problem has a wide variety of use cases. There has been a lot of work done in this area, with the development of elegant tools such as TALOS [5], REGAL [6], and SCYTHE [7]. The ground-truth query is not available in QRE, due to which the output query $\mathcal{Q}_C$ is organically dependent on the specific ($\mathcal{D}_I$, $\mathcal{R}_I$) instance provided by the user.

A variant of the QRE problem was recently introduced in [1], where a ground-truth query is additionally available in hidden form. This problem is termed Hidden Query Extraction (HQE). HQE problem is described in the introduction section. The output query now becomes independent of the initial ($\mathcal{D}_I$, $\mathcal{R}_I$) instance. Our work is enhancing the scope and efficiency of hidden query extraction. We extended the scope of HQE by including the Not Equal predicates into its extractable domain.

Chapter 11

Conclusion and Future Work

We can now extract the hidden queries containing *Not Equal* and *Not Like operators* under EQC^N assumptions. Experiments are performed to verify the proposed solution on complex TPC-H-based queries. We have implemented the suggested optimizations in the UNMASQUE tool's Minimizator and Result Comparator module. We minimized the extraction overhead of UNMASQUE by orders of magnitude. We can extract the *NEP* hidden queries efficiently and accurately.

There are some operators that cannot be extracted by UNMASQUE yet. One possible direction for future work would be to come up with new ideas to extract the filter predicate of the type *column* <> *column*. These types of extraction come under Algebraic predicates extraction. The extraction of nested queries and MINUS set operator is also out of the extractable domain.

References

[1] K. Khurana and J. Haritsa. Shedding Light on Opaque Application Queries. *Proc. of ACM SIGMOD Intl. Conf. on Management of Data, Xi'an, China, June 2021.* ii, 1, 5, 29

[2] K. Khurana and J. Haritsa. 2021. Opaque Query Extraction. Technical Report. Indian Institute of Science. https://dsl.cds.iisc.ac.in/publications/report/TR/TR-2021-02.pdf 22, 23

[3] TPC-DS. http://www.tpc.org/tpcds/ 2

[4] TPC-H. http://www.tpc.org/tpch/ 2

[5] Q. Tran, C. Chan, and S. Parthasarathy. 2014. Query Reverse Engineering. *The VLDB Journal 23, 5 (2014).* 29

[6] W. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava. 2018. REGAL+: Reverse Engineering SPJA Queries. *PVLDB 11, 12 (2018).* 29

[7] C. Wang, A.Cheung, and R. Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proc. of PLDI Conf.* 29

Appendix

TPC-H Based Queries

Q.1:

Select $l\_returnflag$, $l\_linestatus$, sum($l\_quantity$) as sum_qty, sum($l\_extendedprice$) as sum_base_price, sum($l\_discount$) as sum_disc_price, sum(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' and $l\_extendedprice$ <> 33203.72
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' and p_mfgr <> 'Manufacturer#5'
Order by $s\_acctbal$ desc, n_name, s_name, $p\_partkey$
Limit 100;

Q.3:

Select $l\_orderkey$, sum($l\_discount$) 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 $o\_orderdate$ <'1995-03-15' and $l\_shipdate$ >'1995-03-15'
Group by $l\_orderkey$, $o\_orderdate$, $o\_shippriority$
Order by $revenue$ desc, $o\_orderdate$, $l\_orderkey$

Limit 10;

Q.4:
Select *o_orderdate*, *o_orderpriority*, count(*) as *order_count*
From *orders*
Where *o_orderdate* <'1997-09-01' and *o_orderdate* ≥ date '1997-07-01' and *o_orderdate* <>'1997-07-01'
Group By *o_orderdate*, *o_orderpriority*
Order By *o_orderpriority*
Limit 10;

Q.5:
Select *n_name*, sum(*l_extendedprice*) as *revenue*
From *customer*, *orders*, *lineitem*, *supplier*, *nation*, *region*
Where *c_custkey* = *o_custkey* and *l_orderkey* = *o_orderkey* and *l_suppkey* = *s_suppkey* and *c_nationkey* = *s_nationkey* and *s_nationkey* = *n_nationkey* and *n_regionkey* = *r_regionkey* and *r_name* = 'MIDDLE EAST' and *o_orderdate* ≥ date '1994-01-01' and *n_name* not like '%IRAN%'
Group By *n_name*
Order By *revenue* desc
Limit 100;

Q.6:
Select *l_shipmode*, sum(*l_extendedprice*) as *revenue*
From *lineitem*
Where *l_shipdate* ≥ date '1994-01-01' and *l_shipdate* <date '1994-01-01' and *l_quantity* < 24 and *l_shipmode* not like '%AIR%' and *l_shipdate* <>'1994-01-02'
Group By *l_shipmode*
Limit 100;

Q.10:
Select *c_name*, sum(*l_extendedprice*) 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* ≥ date '1994-

01-01' and *o_orderdate* < date '1994-01-01' + interval '3' month and *l_returnflag* = 'R' and *c_nationkey* = *n_nationkey* and *c_name* <> 'Customer#000100867'
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_availqty*) as *value*
From *partsupp*, *supplier*, *nation*
Where *ps_suppkey* = *s_suppkey* and *s_nationkey* = *n_nationkey* and *n_name* = 'ARGENTINA' and *ps_comment* not like '%regular%dependencies%' and *s_acctbal* <> 449.54
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* not like 'SMALL PLATED%' and *p_size* ≥ 4
Group By *p_brand*, *p_type*, *p_size*
Order By *supplier_cnt* desc, *p_brand*, *p_type*, *p_size*;

Q.17:
Select AVG(*l_extendedprice*) as *avgTOTAL*
From *lineitem*, *part*
Where *p_partkey* = *l_partkey* and *p_brand* = 'Brand#52' and *l_shipdate* <> '1994-05-29';

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* and *c_name* <> 'Customer#000060217'
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*, $lineiteml_1$, *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;