Unmasque2: Performance Optimization and Scope Enhancement in Unmasque

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

BY

Sneha Wadekar

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

July, 2023

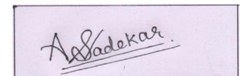
Declaration of Originality

I, **Sneha Wadekar**, with SR No. **04-04-00-10-51-21-1-19437** hereby declare that the material presented in the thesis titled

Unmasque2: Performance Optimization and Scope Enhancement in Unmasque

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2021-2023**.
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: July, 2023 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

© Sneha Wadekar
July, 2023
All rights reserved

DEDICATED TO

My Family

for their love and support

Acknowledgements

I would like to express gratitude to Prof. Jayant R. Haritsa, My Project Advisor and Mentor. I am thankful to him for his invaluable guidance and support throughout my thesis. I am truly grateful for the opportunity to work under his mentor ship. His expertise and encouragement have been instrumental in shaping my academic journey.

I am thankful to Anupam Sanghi and Manish Kesarwani for their valuable input. I am also thankful to Mukul, Aman, and Abhinav for mentoring and assisting me on numerous occasions. I would also like to thank all my lab mates for their constant support.

Finally, I would like to express my heartfelt gratitude to my parents and brother for their unwavering support throughout my life. Their constant presence and encouragement have been invaluable to me, and I am truly grateful.

Abstract

Unmasque (in this thesis called Unmasque 1) is a platform-independent hidden query extractor designed to solve the HQE problem. In this paper, we introduce Unmasque 1.5, an improved version of Unmasque 1 with enhanced performance. Unmasque 1.5 utilizes correlated sampling and view-based halving techniques to replace the old copy-based recursive halving minimizer. It also introduces an efficient hash-based comparator for result comparison. The results section provides a detailed evaluation of the extraction times for both Unmasque 1 and Unmasque 1.5.

Furthermore, we introduce Unmasque 2, a scope-enhanced version of Unmasque 1.5. Unmasque 2 incorporates the performance enhancements and expands the extractable scope to include algebraic predicates, disjunction predicates, outer joins, and not-equal predicates. These additions were previously beyond the extractable domain. We delve into a thorough discussion of these features and present a study on the extraction time of queries involving these predicates.

Contents

Acknowledgements **i**

Abstract **ii**

Contents **iii**

List of Figures **v**

List of Tables **vi**

1 Introduction **1**

1.1 Motivation . . . 2

1.1.1 Performance Optimizations. . . . 2

1.1.2 Scope Enhancements. . . . 2

1.2 Technical Challenges 3

1.3 Our Contribution 4

1.4 Performance Evaluation 4

1.5 Organization 4

2 Unmasque 1: Background **5**

3 Unmasque 1.5:
Performance Optimization. **7**

3.1 Correlated Sampling. . . . 7

3.2 View Based Minimizer. . . . 11

3.3 Hash-Based Result Comparator. . . . 12

4 UNMASQUE 2 :
U1.5 + Scope Enhancements **14**

4.1 Updated Extractable Query Class . . . 15
4.2 Extend Database Minimizer: . . . 16
4.3 Algebraic Predicate Extractor. . . . 17
4.3.1 Active Predicate Extraction. . . . 18
4.3.2 Dormant Predicate Extraction. . . . 18
4.4 Disjunction Extractor. . . . 19

5 Outer Join Extractor. **21**
5.1 Outer Join Existence Checker. . . . 22
5.2 Assumptions. . . . 23
5.3 Create Inner Join Graph. . . . 23
5.4 Refinement of join type on Single Edge Join Graph. . . . 24
5.5 Refinement of Join Type on Multi Edge Join Graph. . . . 26
5.6 Formulate all possible Nesting Sequence of Joins. . . . 28
5.7 Assignment of filter predicates to ON/WHERE clause. . . . 29
5.8 Formulate queries and Eliminate semantically non-equivalent queries (to Q_H): . 32
5.9 Algorithm. . . . 32

6 Not Equal Predicates and Correctness **35**
6.1 Not Equal Predicates. . . . 35
6.2 Correctness. . . . 35

7 Experiments **39**
7.1 U1 v/s U1.5 Performance Enhancement Results . . . 39
7.2 Unmasque 2 . . . 40

8 Related Work **41**

9 Conclusion and Future Work **42**

Bibliography **43**

Appendix **44**

List of Figures

1.1 Unmasque 2 Exemplar Query 3

2.1 Unmasque 1: Extraction Pipeline 5
2.2 List of symbols in the pipeline. 6

4.1 Unmasque 2: Extraction Pipeline 14

5.1 Outer Join + NEP existence checker. 22
5.2 Outer Join + NEP existence checker. 22
5.3 Outer Join Table. 25
5.4 Outer join Graphs 26
5.5 Outer join Graph of U2 Exemplar query. 27
5.6 Example 28
5.7 Multi Edge Join graph 29
5.8 31

7.1 Extraction time Unmasque 1 v/s Unmasque 1.5 39
7.2 Result comparator time 40
7.3 Unmasque 2: Extraction time. 40

List of Tables

Chapter 1

Introduction

Hidden-Query Extraction(HQE) problem aims to identify the Hidden Query while having access to the hidden executable. Formally defined, Hidden-Query Extraction(HQE) is: "Given a black-box application A, containing a Hidden query Q_H (in either SQL format or its imperative equivalent), and a database instance D_I on which Q_H produces a populated result R_I, unmask Q_H to reveal the original query (in SQL format)"[1].

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

Another possible scenario is that the application itself is visible, but its inner workings are still difficult to understand. This can happen in two ways. The first scenario involves the application's construction using intricate SQL queries, which can pose difficulties in understanding. These queries might be automatically generated through machine-generated object-relational mappings, further complicating interpretation for human developers. Second, the application may consist of poorly documented imperative code that is not easily decipherable. This situation can arise when software is inherited from external developers who did not adequately document their code, making it challenging for subsequent developers to understand and modify the application.

The objective of HQE is to uncover the hidden query Q_H and reveal its original form in SQL format. In HQE, the goal is to precisely identify Q_H such that for every instance i, applying Q_H to the database D_i will yield the exact result R_i. In other words, The goal of Hidden Query Extraction is to find the precise Q_H such that $\forall_i Q_H(D_i) = R_i$.

HQE has a variety of use cases such as Imperative Code to SQL Translation, Debugging Applications with stored SQL procedures, Recovering lost source code, etc.

UNMASQUE (Unified Non-invasive MAchine for Sql QUery Extraction) is the first step towards addressing the HQE problem. It is a platform-independent hidden query extractor introduced in [1]. We call it Unmasque 1 or U1 in this thesis. It extracts the hidden query Q_H through "active learning". U1 utilizes the outputs of hidden query executions on carefully crafted database instances to expose the hidden query Q_H. It achieves this through a combination of database mutation and synthetic database generation, without requiring invasive modifications to the application software or underlying database engine.

1.1 Motivation

There are several key motivations that have driven this work. These include:

1.1.1 Performance Optimizations.

Performance Optimization of Unmasque is crucial for achieving faster extraction, efficient resource utilization, cost saving, etc. By optimizing algorithms in Unmasque, we can reduce the extraction time significantly. Correlated Sampling and View-based Minimization techniques can be employed to achieve significant performance optimization at the database minimizer stage.

By employing **Correlated Sampling**, a subset of tuples of the tables can be selected in a way that maintains the join relationship between the tuples, enabling higher chances of a non-empty result when Q_H is run on the sampled database. **View-based Minimization** leverages the creation of logical views instead of actual copies of the minimized database.

To optimize the comparison of results, we employ a **Hash-Based Result Comparator** instead of relying on a comparison-based result comparator.

1.1.2 Scope Enhancements.

Unmasque1 cannot handle algebraic predicates, disjunction predicates, outer joins(referred to as OJ), and not equal predicates(referred to as NEP), we are motivated to develop and implement new extraction components to address some of these shortcomings. Additionally, we also aimed to integrate existing components with the newly designed ones to create an enhanced version of Unmasque, with a wider scope, known as **Unmasque 2**.

The Exemplar Query can be seen in Figure 1.1. It is extractable by Unmasque 2. It includes algebraic predicates, Disjunction Clauses, Outer joins, and Not equal predicates (NEP).

```
SELECT l_suppkey, l_returnflag , p_partkey, l_quantity, ps_availqty, sum(p_size)
FROM lineitem
INNER JOIN partsupp ps ON l_suppkey = ps_suppkey
LEFT OUTER JOIN part ON p_partkey = ps_partkey and ( p_size > 49 OR ps_availqty >
9900 )
WHERE l_shipmode IN ('MAIL', 'SHIP', 'TRUCK') AND l_quantity<>36  AND
(l_quantity >= 30)  AND l_commitdate <= l_receiptdate AND l_returnflag NOT IN ('N')
GROUP BY l_suppkey, l_returnflag , p_partkey, l_quantity, ps_availqty
ORDER BY ps_availqty
LIMIT 100;
```

Figure 1.1: Unmasque 2 Exemplar Query

1.2 Technical Challenges

Debugging and modifying the implementation of the algebraic predicate extractor was necessary to replace the filter predicate extractor module. This change involved ensuring that the algebraic predicate extractor could effectively perform the required extraction tasks.

Additionally, an implementation for the dormant predicates extractor was needed since it was missing from the existing implementation. This involved developing and integrating the functionality to identify and extract dormant predicates that were not previously handled by the implementation.

Both the disjunction and NEP extractor modules demanded meticulous debugging, testing, and modifications to the implementation. These modules were crucial for extracting and processing disjunctions (logical OR conditions) and NEPs within the application. Ensuring their seamless integration into the application required significant time and effort to guarantee the accurate extraction and processing of these types of predicates.

Another challenge was to distinguish between the existence of Not equal predicate or outer joins or both in the hidden query.

Extending the extraction scope to include outer joins presented a challenge for the existing database minimizer, as it became incapable of correctly reducing the database. Incorporating outer joins into the extraction scope required modifications to the database minimizer's algorithms and logic. The inclusion of outer joins in the extraction scope significantly challenged the existing database minimizer and necessitated substantial adjustments to its functionality to ensure accurate and effective reduction of the database.

1.3 Our Contribution

Our contributions encompass several significant enhancements to the system:

1. Correlated Sampling: correlated sampling technique is aimed at enhancing the efficiency of database minimization processes. Recognizing the join relationships between tables, we have implemented a sampling technique that maximizes the probability of hidden queries producing non-empty results by considering the join relationships in the table.

2. View-Based Minimizer: This technique focuses on minimizing the size of the sampled database to a single-row database using systems tuple identifiers to create views.

3. Outer Join Extractor: We developed an outer join extractor module that identifies and extracts outer join operations within the database. This module enables the system to extract outer joins accurately.

4. Integration of Predicate Extractors: We successfully integrated various predicate extractors into the system, including algebraic predicate extractors, disjunction extractors, and Not Equal Predicate (NEP) extractors. These modules enable the system to extract and process complex predicates, enhancing the system's querying capabilities and supporting advanced data analysis tasks.

Overall, our contributions encompass correlated sampling techniques, a view-based minimizer implementation, an outer join extractor, and the integration of various predicate extractors, enhancing extraction scope.

1.4 Performance Evaluation

We have evaluated Unmasque 2 extraction behavior on complex SQL queries containing algebraic predicates disjunctions, outer joins, and not equal predicates arising in a synthetic TPC-H environment. These complex queries are derived from the popular TPC-H benchmark. Our experiments indicate that the hidden queries are precisely identified.

1.5 Organization

The organization of the following chapters is as follows. Chapter 2 gives a very brief description of Unmasque. Chapter 3 discusses the performance optimizations and their algorithms. Starting from Chapter 4 we go through each scope enhancement. Chapter 5 discusses the extraction process of outer join and its algorithm. Chapter 6 discusses the Not equal predicate extractor. Chapter 7 goes through the experimental evaluation. Chapter 8 discusses the related work. chapter 9 concludes the thesis and discusses some possible upcoming works in unmasque. The appendix contains some experimental test queries.

Chapter 2

Unmasque 1: Background

Unmasque 1 is capable of extracting a basic set of warehouse queries that include the core SPJGAOL clauses, which refer to single-block equijoin queries with conjunctive predicates.

The class of queries that UNMASQUE can handle is defined in [1] as Extractable Query Class (EQC). Unmasque includes higher-order logic constructs such as comparators and multi-linear scalar functions. To achieve this broader coverage, certain assumptions are made in [1]. The supported queries in this framework are referred to as Extractable Query Class (EQC).

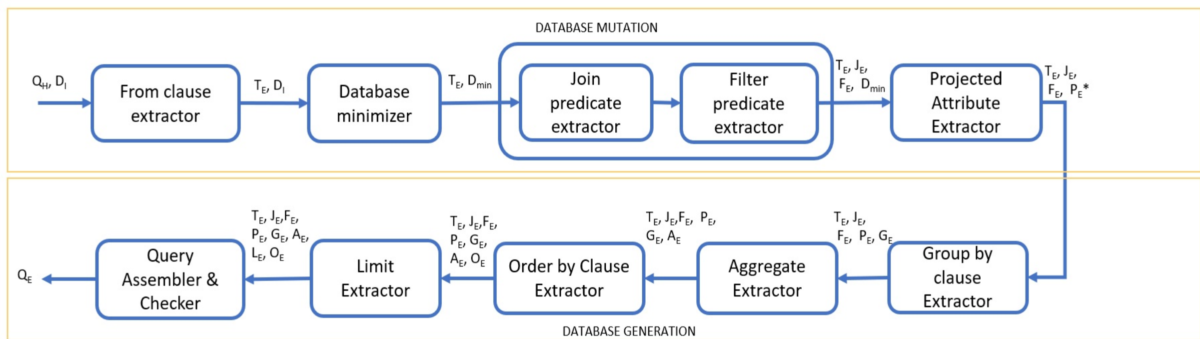


Figure 2.1: Unmasque 1: Extraction Pipeline

The Unmasque 1 pipeline consists of several steps that contribute to the process of hidden query extraction. Here is a description of the Unmasque 1 pipeline:

1. The from clause extractor retrieves tables in the hidden query.

2. The database minimizer uses a copy-based recursive halver to reduce the initial input database to a minimized database with a single row database called D_1.

3. The join predicate extractor extracts inner joins.

4. The filter predicate extractor is mainly able to extract arithmetic predicates. These predicates are on only non-key columns and are of the type *column op value*. Further, for numeric columns, $op \in \{=, <, <=, >, >=, between\}$. Whereas for textual columns, $op \in \{=, like\}$.

5. The Project, Group By, Aggregate, Order By, and Limit are extracted as discussed in [2].

6. The query assembler formulates the query and the result comparator/ checker is a comparison-based technique to check the results of the hidden query and extracted query on the initial database.

Symbol	Meaning	Symbol	Meaning
Q_H	Hidden Query	G_E	Set of group by column
D_I	Initial database instance	A_E	Set of Aggregations with mapped result columns
D_{min}	Minimized database	L_E	Limit value
T_E	Set of tables in the query	O_E	Sequence of ordered result columns
J_E	Set of join predicates	Q_E	Extracted query
F_E	Set of filter predicates	P_E	Set of projections
P_E*	Set of projection + unidentified Aggregations.	NEP_E*	NEP predicate extracted in current loop's iteration.
Q_E*	Partially correct extracted query	D_I*	Subset of initial database.
F_E*	New Filter predicates from D_{min}*	D_{min}*	Minimization of D_I*

Figure 2.2: List of symbols in the pipeline.

Chapter 3

Unmasque 1.5: Performance Optimization.

Unmasque 1.5 (also addressed as U1.5) is the performance-optimized version of U1. The sampling technique described in section 3.1 is employed to create an initial database sample. Subsequently, the sampled database undergoes a recursive minimization process, utilizing the view-based recursive halving technique outlined in section 3.2. This process culminates in the generation of a single-row database, denoted as D_1.

3.1 Correlated Sampling.

Correlated sampling (CS2) is developed by storing correlated sample tuples to preserve the joined relationships. That is, tuples of a relation are included in the sample of a relation if they join with already drawn sample tuples of other relations[7]. Using the already sampled tuples and indexes, we can draw new tuples which are included in the sample of the next table.

First, we perform a reduction of the schematic join graph. Here's how the reduction process works:

1. The input is the complete join graph, including all tables and their relationships. And Q_H's From clause which specifies the tables of focus.

2. Selecting Relevant Tables: From the complete join graph, only the tables mentioned in the From clause are selected. Other tables are excluded from further consideration.

3. Eliminating Irrelevant Edges: The edges in the join graph that connect the irrelevant tables (not present in the FROM clause) are eliminated.

The result is a **simplified join graph** that contains only the relevant tables and their join relationships as specified in the From clause. This reduced graph provides a focused representation of the tables and their connections that are necessary to perform minimization.

The next procedure is as shown in Algorithm 1. The *seed_sample_rate* refers to the percentage of tuples that will be included in the sample. To update the seed sample rate, we follow a geometric progression. *global_join_graph* is the simplified join graph we discussed. *global_core_sizes* is a dictionary storing tables in From clause and their sizes. $CheckNonEmpty()$ Returns True if the $Q_H(D_{sampled})$ has non empty results, Else returns False.

The Algorithm 1 is a procedure that performs sampling on a global join graph based on a given seed sample rate. Here's a breakdown of the steps in the algorithm:

1. Initialize variables:
   (a) *iterations*: the number of iterations (n)
   (b) *seed_sample_rate*: the initial seed sample rate (rate)
   (c) *scaling_factor*: a constant factor for updating the seed sample rate.
   (d) $CS2_flag$: a flag indicating whether CS2 sampling has passed.
   (e) *sampleStatus*: a data structure to track the sampling status of tables.(boolean value)
2. While there are iterations remaining:
   (a) If CS2 flag is False:
      i. Update the seed sample rate by multiplying it with the scaling factor. Decrease the number of iterations by 1.
      ii. For each table in the vertices of the global join graph with $sampleStatus = False$. Find the table with the maximum global core size (*max_cs*) and $sampleStatus == False$. Set the base table to the table with the maximum core size. Sample the base table using the seed sample rate. Update the *sampleStatus* of the base table to $True$. Sample the remaining tables connected to the base table (*table_name*) using the sampled base table. Update their *sampleStatus* to True.
      iii. Check if the result of Q_H on sampled database is non-empty by invoking the $CheckNonEmpty()$ function. If the result of Q_H on sampled database is empty : Revert the sampling and restore the original database and Set the $CS2_flag$ to $False$. If the result of Q_H on sampled database is non-empty : Set the $CS2_flag$ to $True$ (sampling has passed)

iv. If $CS2_flag$ is True: Save the sampled database. Return from the procedure (sampling is successful).

v. If $CS2_flag$ is False continue the while loop.

(b) If $CS2_flag$ is still False after exiting the while loop: CS2 sampling failed for all iterations. Return from the procedure.

The algorithm terminates either when sampling is successful or when it fails for all iterations.

Algorithm 1 Correlated Sampling

$iterations \leftarrow n$
$seed_sample_rate \leftarrow rate$
$scaling_factor \leftarrow x$
$CS2_flag \leftarrow False$
$sampleStatus \leftarrow \{\}$
while $iterations > 0$ **do**
  if $CS2_flag == False$ **then**
    $seed_sample_rate* = scaling_factor$
    $iterations = iterations - 1$
    for *table_name in vertices of (global_join_graph with sampleStatue == False)* **do**
      $max_cs \leftarrow 0$
      for *i in vertices of (global_join_graph with sampleStatue == False)* **do**
        if $max_cs < global_core_sizes[i]$ **then**
          $base_table = i$
          $max_cs = global_core_sizes[i]$
        end
      end
      Sample the *base_table* using *seed_sample_rate*
      sampleStatus[*base_table*] = True
      Sample remaining tables connected to base table *table_name* using the sampled base table and *global_join_graph*. And update their *sampleStatus* to True.
    end
    if $CheckNonEmpty() == False$ **then**
      revert the sampling, restore the database $CS2_flag = False$
    else

    end
    $CS2_flag = True$
  else
    CS2 sampling PASSED.
    Save the Sampled database.
    return
  end
end
CS2 failed for all iterations
return

3.2 View Based Minimizer.

Once Correlated Sampling is completed, the sampled tables undergo a minimization stage known as View-Based (recursive halving) minimization shown in Algorithm 2. This technique focuses on minimizing the size of the sampled database to a single-row database using systems tuple identifiers to create views.

A tuple identifier represents a physical location of a row. 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 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. Using *ctid*'s, we can quickly locate the required chunk of large Tables.

We create a view, having the upper half of the table. We query the view by hidden query Q_H. If the results of Q_H on the minimized database are non-empty, we proceed with the upper half. Then recursively consider the upper and lower half of this half. If the results of Q_H on the minimized database are empty, we proceed with the lower half of the database and recursively consider its upper and lower half. The process of recursive halving using views is then followed until we reach a single row database called D_1. View creation is a constant-time operation. The extra time and space of materializing the table is avoided. Algorithm 2 shows the algorithm for view-based minimizer.

Algorithm 2 View-Based Minimizer

$max_no_of_rows = 1$

for *table_name in (global_core_relations)* **do**

  Retrieve the *start_ctid* values of the first tuple of the table.

  Initialize *start_page* with the page numbers.

  while $global_core_sizes[table_name] > max_no_of_rows$ **do**

    Calculate the *mid_page* using *start_page* and the number of tuples fitting in a single page.

    Create view using *start_ctid* & *mid_ctid*.

    if *the result of* Q_H *on the minimized database is non-empty by executing* $CheckNonEmpty()$ **then**

      Consider tuples from *start_ctid* to *mid_ctid* for further minimization

    else

      Consider tuples from *mid_ctid* for further minimization

    end

  end

end

3.3 Hash-Based Result Comparator.

A hash-based result comparator is a technique used to compare and match the results of two or more queries based on the hash values of the results. The Row Hash Method is a technique used to calculate a hash value for a table by applying a hash function to each individual row or tuple in the table and aggregating the results. The query we use is: Select sum(hashtext) from (select hashtext(r_e::TEXT) FROM r_e) as T. The provided SQL query performs a calculation using the Row Hash Method. Let's break it down step by step:

1. ($select\ hashtext(r_e :: TEXT)\ FROM\ r_e$) is an inner query that selects the hashtext value of each tuple in the r_e table. The $r_e :: TEXT$ part converts the tuple into a string representation to generate a hash.

2. *as T* assigns the result of the inner query to the alias T, creating a temporary table.

3. Finally, the outer query $select\ sum(hashtext)\ from\ T$ calculates the sum of the *hashtext* values from the temporary table T. This aggregation represents the checksum value of the r_e table.

The purpose of this query is to compute the checksum value of the r_e table using the Row Hash Method. By summing the individual hash values, you obtain a single value that serves

as a unique identifier for the table's contents. This checksum value can be used for comparison with other tables or as a measure to detect changes or equivalence between different instances of the r_e table.

Chapter 4

UNMASQUE 2 : U1.5 + Scope Enhancements

The new extraction pipeline of Unmasque 2 is shown in Figure 4.1. It includes the performance optimizations discussed in Chapter 3. In addition to that, it also consists of the additional extraction components that contribute to scope enhancements. It is capable of extracting correctly the exemplar query of Unmasque2(figure 1.1).

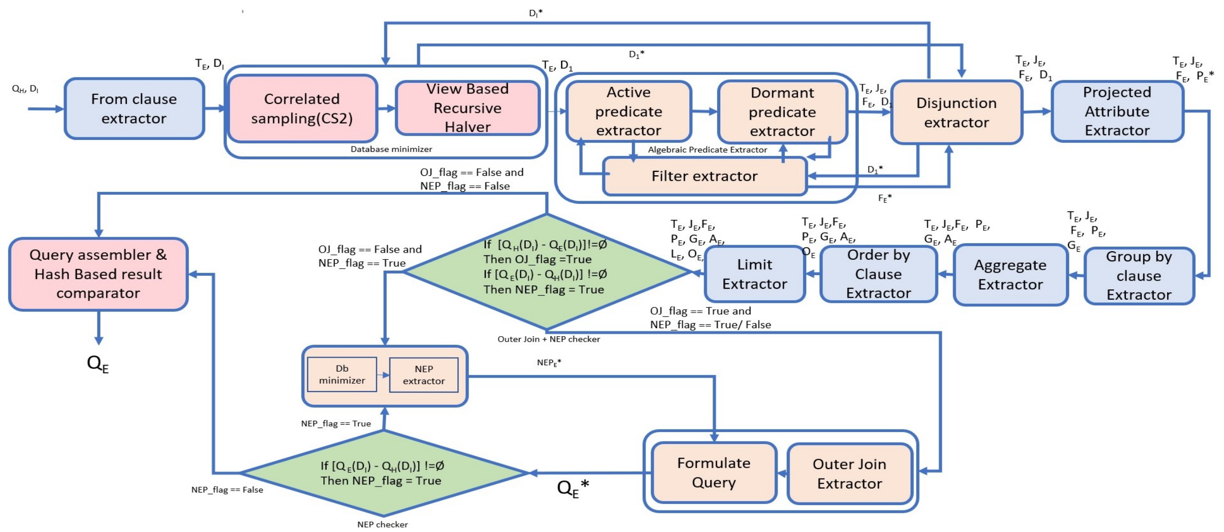


Figure 4.1: Unmasque 2: Extraction Pipeline

The walk-through of the Unmasque 2 extraction pipeline is as follows:

1. Identify the tables involved in the query, which are referred to as "core relations".

2. The database minimizer, consists of two stages. Correlated sampling is used to reduce the cardinality of the database by employing a sampling technique. The resulting smaller database is then recursively minimized using the View-based minimizer. The final outcome is a single-row minimized database, denoted as D_1.

3. Algebraic predicate extraction is performed on D_1.

4. The Disjunction predicate extractor identifies any disjunctions in the query.

5. The pipeline proceeds with the extraction of Projection, Group By, Aggregate, Order By, and Limit operations[1].

6. After the limit extractor, the pipeline checks for the presence of Outer Joins and Not Equal Predicate (NEP) in the Q_H.

7. If an outer join is detected, extraction enters the Outer Join extractor module.

8. Following the outer join extraction, if NEP is present, the pipeline proceeds to the NEP extractor loop.

9. The pipeline concludes with the query assembler combining the extracted components to form the final query, which is then compared using a hash-based result comparator.

Hidden queries need to fulfill certain criteria. Many predicates do satisfy these assumptions inherently.

In the upcoming sections, we discuss some changes done in some modules along with the significant advancements and enhancements in scope that have been achieved in Unmasque 2.

4.1 Updated Extractable Query Class

The limitation in [1], which says that, Filter predicates feature only non-key columns and are of the type column op value. Further, for numeric columns, $op \in \{=, <=, <, >, >=, between\}$, whereas for textual columns, $op \in \{=, like\}$, has been eliminated. Unmasque 2 can extract filter predicates if they are of the type $\langle$Column op X$\rangle$ where X can be a column or value and $op \in \{=, <=, <, >, >=, between\}$, whereas for textual columns, $op \in \{=, like\}$. In addition to the previously mentioned operators, it is worth noting that the Not Equal operator ($<>$) and disjunction predicates were not extractable in the earlier domain. However, they have now been made extractable.

1. Filter predicates are of the type $\langle$Column op X$\rangle$ where X can be a column or value and $op \in \{=, <=, <, >, >=, between\}$, whereas for textual columns, $op \in =$, like.

2. The Not Equal Predicate present in the filter is only on the non-key columns and is of the type column op value where $op \in \{<>, notlike\}$.

3. Filter is a conjunction of disjunctions. filter predicates present in disjunction are of type column op value. Further, for numeric columns, $op \in \{=, <=, <, >, >=, between\}$, whereas for textual columns, $op \in =$, like.

4. Every true assignment of filter, such that only one predicate is satisfied from each clause, contributes at least one unique row to the output.

5. Algebraic predicates and NEP predicate must not be a part of disjunction predicates. NEP must not be present in disjunction with another filter predicate. Let s1 = {set of attributes participating in Algebraic predicates}. s2 ={attributes in disjunction predicates}. s3= { attributed having Not Equal predicate }. Then $s1$ intersection $s2$ intersection $s3$ must be empty.

6. To extract Outer Joins, the next three prerequisites are required to be followed. These are generally followed by benchmark databases and queries : Complete non-null database: The initial database (D_I) must be a fully populated database without any $NULL$ values.

7. At least one Non-aggregated attribute in projection: The projection clause of the query should include at least one non-aggregated attribute from each of the tables involved in the join operation.

8. Tuple with all Non-null values in result of $Q_H(D_I)$: The outcome of the hidden query (Q_H) on the initial input database (D_I) must have at least one tuple where all the projected attributes have non-null values.

We share the same basic structural restrictions. However, our extraction scope is significantly enlarged to include additional predicates and functionalities. . We hereafter refer to this class of supported queries as Extractable Query Class - Unmasque 2 EQC_{U2}).

4.2 Extend Database Minimizer:

The database minimizer module in U1 checked if the result of the hidden query on the minimized database gives a non-empty result or not. But to extract Outer Joins, the database minimizer needs to make sure of two conditions: First, the results of hidden-query on minimized tables is not empty. Second, out of the tuples which are part of the non-empty results, at least one tuple must be such that all projected attributes have a non-null value. If we make sure that

the tables are minimized making sure of these two conditions, the outcome of the database minimizer module is a D_1 such that in the results of $Q_H(D_1)$, none of the projected attributes have a null value. This indirectly ensures that if Q_H has outer joins, D_1 will be a single-row database where all join conditions are satisfied.

We update the Correlated Sampling and View minimizer for the above logical change. We need to replace $CheckNonEmpty$ with $CheckNullFree$. The $CheckNullFree$ function will return $True$ if the results of Q_H on the sampled or minimized database have at least one row with all non-null values. We call such a row a Non-null row. Otherwise, it returns $False$.

4.3 Algebraic Predicate Extractor.

We will explore the extraction process for hidden queries that involve Algebraic Predicates. These predicates consist of comparisons between columns, such as *column op column*, where $op \in \{=, <, <=, >, >=\}$ for numeric columns, and $op \in \{=, like\}$ for textual columns. Additionally, we will also consider Arithmetic Predicates, which involve comparisons between a column and a value, such as a column op value, where $op \in \{=, <, <=, >, >=, between\}$ for numeric columns and $op \in \{=, like\}$ for textual columns.

Algebraic Predicates present in Q_H are classified into two types:

1. Dormant Predicates: If there is a predicate $col <= col'$ with col having an upper bound predicate $col <= val$ and col' having a lower bound because of predicate $col' >= val'$, where the value of col in D_1 is less than val' and the value of col' in D_1 is greater than val. Then the predicate $col <= col'$ is inherently satisfied and will have no effect on the result and bounds when a single mutation is performed. Dormant predicates are inactive because they are overshadowed by other predicates and the overlapping bounds between the columns. These predicates do not contribute to the query result because they are already satisfied indirectly.

2. Active Predicates: All remaining predicates that are not classified as Dormant Predicates are considered Active Predicates.

Dynamic nature of predicates: It's important to note that the classification of predicates is based on minimized database D_1. For different D_1, some active predicates may become dormant, and some dormant predicates may become active. However, the classification definition remains consistent in terms of D_1.

4.3.1 Active Predicate Extraction.

We will discuss for the active predicates of type *column op X* where $op \in \{=, <=, <, >=, >\}$ and X can be a column or value. The filter predicate extractor is used to get the bounds on the columns. We will first validate whether the bound is concrete or variable. The Validator does it by manipulating the values (within the bounds) one by one of all such columns whose value in D_1 is equal to the bound. If the bounds turn out to be variable (due to column col), we will iteratively find the new bounds by assigning the col as min or max depending upon the nature of the bound otherwise, we will conclude that the bound is concrete.

4.3.2 Dormant Predicate Extraction.

Dormant Predicates can arise due to the overlapping bounds of two columns present in different components. One column decides a variable bound on the other column but has a value greater than the concrete upper bound of the other column and vice-versa.

l_extendedprice $<=$ *o_totalprice and*
l_extendedprice $<=$ 70000 *and*
o_totalprice $>=$ 60000

Let's consider the above example to further discuss this.

Detection of Dormant Predicates

To check the existence of dormant predicates, we will perform the following steps:

1. Choose a predicate and select one of its columns to assign a minimum value.

2. Keep all other columns at their maximum possible s-value.

3. If the hidden query, based on the mutated D_1, yields empty results, it indicates the presence of a dormant predicate.

For instance, let's consider a case where we choose the predicate involving *o_totalprice*. Assigning *o_totalprice* its minimum value of 60000 and *l_extendedprice* its maximum value i.e. 70000. This leads to an empty result because the predicate *l_extendedprice* $<=$ *o_totalprice* won't be satisfied. And we can conclude that a dormant predicate exists in Q_H.

Extraction of Dormant Predicates

To extract the Dormant predicate we convert it to active predicate. To convert the dormant predicate into an active predicate, we follow these steps:

1. Choose a pair of predicates, such as Predicate 1: $l_extendedprice <= 70000$ and Predicate 2: $o_totalprice >= 60000$.

2. Assign the maximum possible value for all columns, except for the columns in one of the predicates. Say we assign *o_totalprice* its maximum value and the remaining columns their minimum value.

3. During the evaluation of the lower bound of *o_totalprice*, it turns out to be the same as the value of *l_extendedprice*. To validate this, we manipulate the value of *l_extendedprice*. By converting the dormant predicate into an active one, we can properly incorporate its influence on the query evaluation and bounds determination process.

The dormant predicate extractor implementation was not complete previously but has been done now. The algorithm described in [5] was implemented to extract the Dormant predicates. This implementation focuses on identifying dormant predicates in a given Q_H. For details on the Algebraic predicates refer [5]. Once the algebraic predicates have been extracted, the next step is to identify and extract disjunction predicates.

4.4 Disjunction Extractor.

In order to extract any predicate in disjunction, we must have such a minimization in which that particular predicate is the only one from its clause being satisfied. Hence we want the filter to be a conjunction of disjunctions. And every true assignment of filter, such that only one predicate is satisfied from each clause, contributes at least one unique row to the output.

The Disjunction extractor is called after the algebraic predicate extractor module and thus receives algebraic predicate and arithmetic predicates as input, which necessarily contains exactly one predicate from each clause according to [6].

From the exemplar query of Unmasque2, the disjunction clause is: $(\, p_size > 49\; OR\; ps_availqty > 9998)\; and\; l_shipmode\; IN\; ('MAIL','SHIP','TRUCK')$. We are going to consider this as our running example to discuss the extraction of disjunction predicates.

Say $F_E = (p_size > 49\; and\; l_shipmode =' MAIL')$ are the filter clause detected initially. The algorithm makes multiple calls to the database minimizer and Filter Extraction modules, but as shown in the pipeline the calls are both subroutine calls and they transfer the control back to the algorithm.

Let's see the steps in detecting the disjunction clause in the exemplar query:

1. Delete the rows from database having $p_size > 49$ and keep only the rows having $l_shipmode =' MAIL'$

2. Minimize this database and call the filter extractor module.

3. Filter Clause Extractor returns ($ps_availqty > 9998$ *and* $l_shipmode =' MAIL'$) hence n $ps_availqty > 9998$ is added to the Disjunction list of $p_size > 49$.

4. Delete the rows from database having $ps_availqty > 9998$ or having $p_size > 49$ and keep only the rows having $l_shipmode =' MAIL'$.

5. Executable output will be empty here and hence this clause is concluded.

6. In a similar fashion, the second clause will be extracted completely.

Integration of the disjunction extractor was done by me. For any details about the disjunction extractor refer to [6].

Chapter 5

Outer Join Extractor.

A SQL Join is used to combine data from multiple tables to generate a single output table, selecting specific columns from each table. The tables are linked based on one or more common attributes. The **Join condition** specifies the criteria which must be satisfied to combine rows.

- The INNER JOIN joins return rows from both tables as long as the join conditions are met. Only the rows with matching values in both tables are included in the output.
- The LEFT OUTER JOIN returns all rows from the table on the left side of the join and includes matching rows from the table on the right side. If there are left table rows with no corresponding matches on the right side, the space for the right side will be filled with null values.
- The RIGHT OUTER JOIN is akin to the LEFT JOIN but with reversed roles. It returns all rows from the table on the right side of the join and includes matching rows from the table on the left side. Any rows without matching values on the right side will be filled with null values.
- The FULL OUTER JOIN keyword combines the results of both a LEFT JOIN and a RIGHT JOIN. It creates a result that includes all rows from both tables. If rows don't have matches in the other table, null values will be present in the result.

Outer joins are important in relational databases because they allow for the retrieval of data from two or more tables, even if there is no match between the joined columns. They enable the inclusion of unmatched rows in the query results, which can be crucial for various data analysis and reporting scenarios.

Benchmark queries often include outer joins to evaluate the performance and efficiency of database systems. TPC-DS[8] Query 80, Query 78, Query 72, and Query 40 are a few examples of benchmark queries having outer joins.

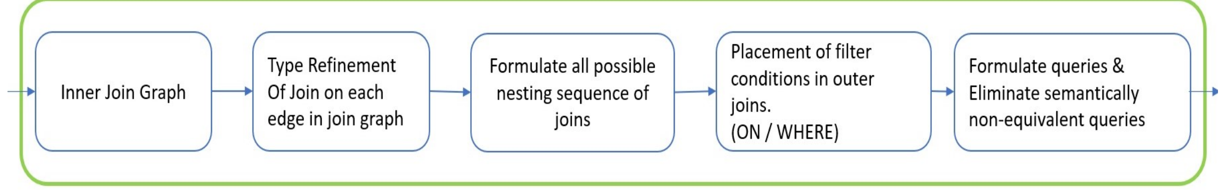


Figure 5.1: Outer Join + NEP existence checker.

5.1 Outer Join Existence Checker.

After extracting the limit clause, we proceed to examine the hidden query for the presence of Outer Joins and NEP (Not Equal Predicate).

- To detect Outer Joins, we compare the results of the Hidden Query i.e. $Q_H(D_I)$ with the results of the extracted query i.e. $Q_E(D_I)$. If the hidden query(Q_H) produces rows that are not present in the extracted query's(Q_E) results, and at least one column in those rows has a $NULL$ value, we identify the presence of outer join predicates.
- To identify the presence of NEP predicates, we compare the results of the extracted query with the results of the hidden query. If the extracted query yields rows that are not found in the hidden query's results, we conclude that NEP predicates exist.

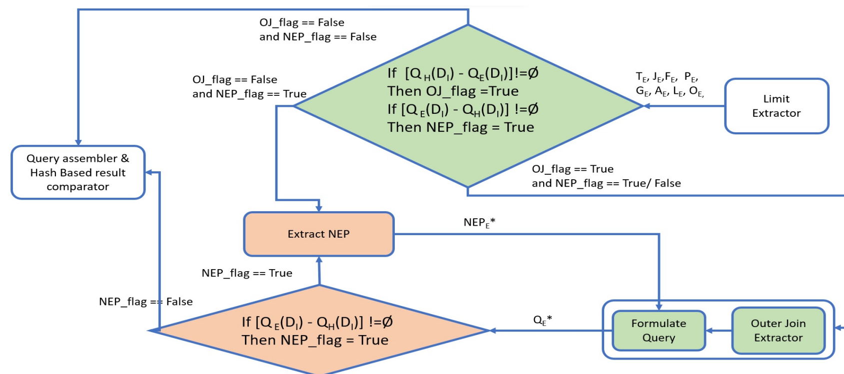


Figure 5.2: Outer Join + NEP existence checker.

At this stage, we are concerned about the presence of Outer Joins, which is detected as shown: If $Q_H(D_I)$ minus $Q_E(D_I)$ = non-empty then, Outer Join Flag = True. Else, Outer Join Flag = False.

5.2 Assumptions.

To extract Outer Joins, the following prerequisites are required to be followed. These are generally followed by benchmark databases and queries.

1. Complete non-null database: The initial database (D_I) must be a fully populated database without any $NULL$ values.

2. At least one Non-aggregated attribute in projection: The projection clause of the query should include at least one non-aggregated attribute from each of the tables involved in the join operation.

3. Tuple with all Non-null values in result of $Q_H(D_I)$: The outcome of the hidden query (Q_H) on the initial input database (D_I) must have at least one tuple where all the projected attributes have non-null values.

These prerequisites ensure that the resulting minimized database (D_1) obtained by the database minimizer will have non-null values for all the projected attributes, enabling the extraction of Outer Joins accurately.

5.3 Create Inner Join Graph.

All Outer Joins in the Hidden Query (Q_H) will be identified as Inner Joins by the previous extracting components in the extraction Pipeline.

Lemma 5.1 *If the Q_H has Outer Joins, then Algebraic Predicate Extractor will detect those Outer Joins as Inner Joins.*

Proof: For ease of explanation, let's assume the hidden query Q_H contains an outer join between two tables *partsupp* and *lineitem*, represented as *partsupp* $(left)$*outer join lineitem ON ps_suppkey* $=$ *l_suppkey*. This same explanation can be generalized for any type of outer join, and any tables/keys combination. It will still hold true.

During the execution of the Algebraic Predicate Extractor, the outer join predicates in Q_H will be identified as equivalent inner join predicates. This is achieved through the following steps:

1. Assumption 3 in section 5.2 states that when executing a hidden query (Q_H) on the initial input database (D_I), the resulting output must contain at least one tuple where all the projected attributes have non-null values. This assumption is important because it ensures that the Database Minimizer will guarantee the satisfaction of the outer join conditions in D_1, where $ps_suppkey = l_suppkey$. We minimize the database to such a D_1.

2. The next stage is getting the filters/bounds on each of the columns using D_1. We get $(lineitem, l_suppkey, 1, 1)(partsupp, ps_suppkey, 1, 1)$. [syntax being followed is :(table name, attribute name, lower bound, upper bound)]

3. For each of the filters extracted, we check in D_1 for the existence of any other column having the same value as the bounds in the filter predicate we are considering. So, we will check in D_1 for attributes having same values as that of bounds on *lineitem.l_suppkey*, i.e. upper bound = lower bound = 1. We detect that *partsupp.ps_suppkey* has same value i.e. 1.

4. We validate for the possibility of the existence of predicate: $ps_suppkey = l_suppkey$. We perform mutations on D_1 to create D_{1mut} by assigning $l_suppkey = ps_suppkey = x$, where x != 1(old value), and x belongs to a feasible range of both attributes. If $Q_H(D_{1mut})$ gives results with at least one non-null row, we extract the filter predicates which will now reflect the mutations. $(lineitem, l_suppkey, x, x)(partsupp, ps_suppkey, x, x)$. Hence, we can conclude $ps_suppkey = l_suppkey$ exists.

5. The from and where clause of the query will look like this: From lineitem, partsupp where $ps_suppkey = l_suppkey$. Which is an equivalent representation of inner join.

Since the Algebraic Predicate Extractor identifies the outer join predicate as an inner join, it follows that all outer join predicates in Q_H have been extracted as inner join predicates in Q_E before we enter the Outer join existence checker module.

Therefore, we can conclude that if Q_H has outer joins, the Algebraic Predicate Extractor will detect those outer joins as inner joins in the extracted query Q_E.

5.4 Refinement of join type on Single Edge Join Graph.

Consider our Q_H has a single join. So the join graph will have a single edge. We require at least one non-aggregated attribute from each table participating in join to be present in

the projection clause. We will observe these attributes, i.e. observe if their values are null or non-null.

Steps to determine the type of join on this edge:

1. Perform mutation on D_1 to generate mutated D_1 represented as D_{1mut}. Break the join condition by negating one of the keys in the condition i.e. perform mutations on D_1 to make the join condition $False$.

2. Run the Q_H on D_{1mut}.

3. Determine the type of join using the table in Figure 5.3

In the table of Figure 5.3 if the entry is Non-Null, it means that for the projected attribute of that table, at least one tuple has a Non-Null value. If the entry is Null, it means that for the projected attribute of that table, all tuples have a Null value.

Projected Attribute of table-x	Projected Attribute of table-y	Type of join
Empty results		Inner Join
Non-Null	Null	Left Outer Join
Non-Null	Non-Null	Full Outer Join

Figure 5.3: Outer Join Table.

Single-edge join graphs will look as shown in Figure 5.4. Sub-figure (a) represents a single edge join graph where join is inner join, notice that the edge is non-directed. Sub-figure (b) represents a single-edge join graph where the left outer join exists. Sub-figure (c) represents a single edge join graph where a full outer join is present. We have not considered the Right outer joins in the above discussion because, they are exactly similar to the left outer joins, but the sequence of tables is reversed. If the hidden query happens to have a "t1 right outer join t2" or "t2 left outer join t1", both are semantically equivalent, and we can identify both cases as "t2 left outer join t1".

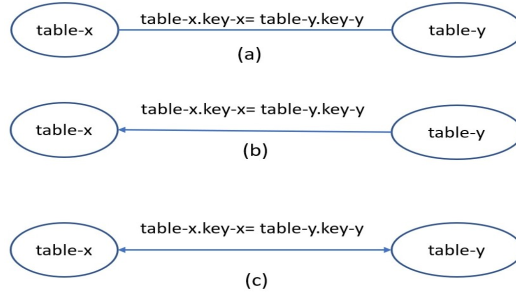


Figure 5.4: Outer join Graphs

5.5 Refinement of Join Type on Multi Edge Join Graph.

When we have a join graph with multiple edges, it means that there are multiple join operations involved in the query. Each edge in the join graph represents a specific join between two tables or relations. The type of join associated with each edge in the join graph can be determined independently of the joins on the other edges.

The arrows at each edge represent the importance of the table at that end of the edge.

Lemma 5.2 *If multiple edges exist in the join graph, then the type of joins corresponding to each edge can be determined independently of the joins on the remaining edges.*

Proof: **Proof by Induction.** Let n be the number of edges in the join graph.
Base case: n=2. It is simple to verify the base case to be true, following the explanation in section 5.4 for each of edge.
Assuming n=k is true.
Assuming, if there are k edges in the join-graph we can determine the type of join for any edge say e, independent of the type of other $k-1$ edges
Showing n=k+1 is true, using this assumption
i.e. If there are $k+1$ edges/joins in join-graph we can determine the type of join for edge e_{k+1} i.e. for the join OJ_{k+1}, independent of other k edges. Suppose Q_H has a join condition of type:

$$T_1\ OJ_1\ T_2 ... T_i\ OJ_i\ T_{i+1} .... T_k\ OJ_k\ T_{k+1}$$

In the join graph, we call the edge corresponding to OJ_i as edge e_i. In D_1 all join conditions will be satisfied. To determine the type of join at edge e_{k+1}. We break the join condition corresponding to OJ_{k+1} to create a mutation of D_1 called D_{1mut}. Then, Run Q_H on D_{1mut}.

Consider the graph before edge e_{k+1}. As the join condition corresponding to all other edges will be satisfied in D_{1mut}, the graph except edge e_{k+1} and table T_{k+1} can be replaced by a single

partial-outcome table i.e. $T_{partial}$. In an environment where the database tables have a single tuple, and the join condition is satisfied by these tuples, the outer joins get reduced to the inner join. Hence in the partial-outcome table ($T_{partial}$) none of the projected attributes will be $NULL$. We can represent the join as:

$$T_{partial}\ XJ_k\ T_{k+1}$$

Using the method described in section 5.4 join at edge e_{k+1} can be determined. Hence the lemma is True.

□

Another way to look at **Lemma 5.2** is that, if there are $k+1$ edges in join-graph we can determine the type of join for the edge e_i, independent of the remaining k edges. suppose hidden-query has a join condition of type as shown in lemma 5.1.

To determine the type of join at edge e_i. Break the join condition corresponding to the XJ_i by negating any one of the keys in the join condition. this will make the join condition false in D_{1mut}

Run Q_H on mutated D_{1mut}. As the join condition corresponding to all other edges will remain satisfied, the graph before edge e_i can be reduced to a partial outcome table- $T_{partial1}$. Similarly, the graph after edge e_i can be reduced to a partial outcome table- $T_{partial2}$.these partial outcome tables will not have any $NULL$ values, because all other join conditions are still being satisfied by D_{1mut}. The partial outcome join looks like this:

$$T_{partial1}\ XJ_i\ T_{partial2}$$

$T_{partial1}$ and $T_{partial2}$ will not have $NULL$ values. So by following section 5.4 join at edge e_i can be determined.

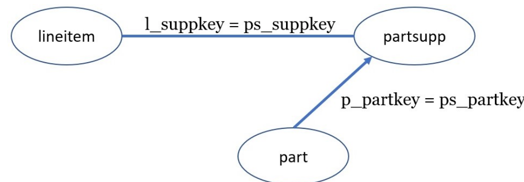
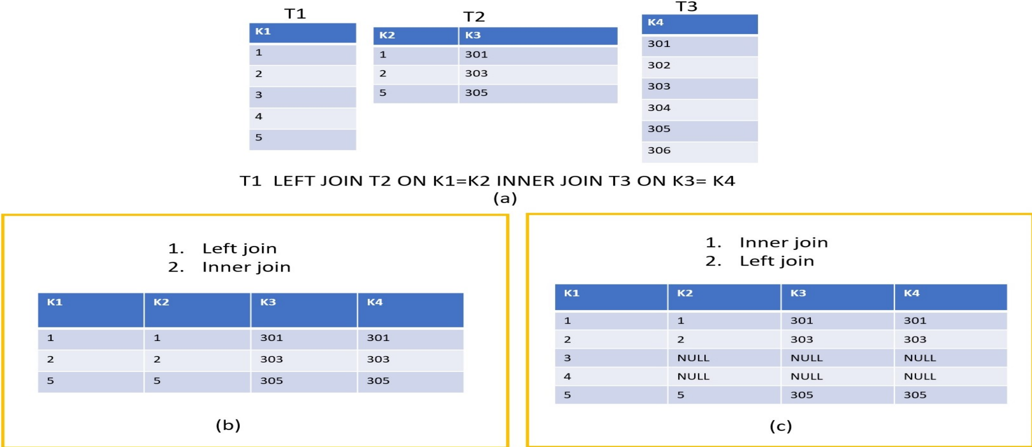


Figure 5.5: Outer join Graph of U2 Exemplar query.

5.6 Formulate all possible Nesting Sequence of Joins.

There is a logical nesting in the query itself. Outer Joins are not commutative like Inner Joins. So the possible Nesting Sequence on the joins must be determined.

Figure 11(a) shows the three tables T1, T2, and T3. Figure 11(b) shows the results of "T1 LEFT JOIN T2 ON K1=K2 INNER JOIN T3 ON K3= K4" when the left outer join is performed first and inner join is performed second. Figure 11(c) shows results when the inner join is performed first and the left outer join is performed second. The results of the two nesting sequences can be observed to be different:



T1

K1
1
2
3
4
5

T2

K2	K3
1	301
2	303
5	305

T3

K4
301
302
303
304
305
306

T1 LEFT JOIN T2 ON K1=K2 INNER JOIN T3 ON K3= K4
(a)

1. Left join
2. Inner join

K1	K2	K3	K4
1	1	301	301
2	2	303	303
5	5	305	305

(b)

1. Inner join
2. Left join

K1	K2	K3	K4
1	1	301	301
2	2	303	303
3	NULL	NULL	NULL
4	NULL	NULL	NULL
5	5	305	305

(c)

Figure 5.6: Example

Since these two nesting sequences give different results. We need to determine the nesting sequence in Q_H.

All possible permutations can always be an option to check, but we can highly optimize the way we determine the sequence of nesting of joins (or joins) in a join graph. By doing a Combinatorial enumeration of all feasible joins. This is in principle exponential but we have tried to minimize the impact by doing some localized optimizations. By following the method described.

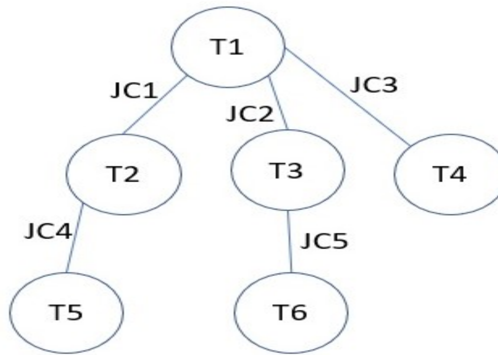


Figure 5.7: Multi Edge Join graph

Figure 5.7 is a join graph representing it as non-directed for ease of understanding. Consider each edge as the root of the tree. We formulate a possible nesting sequence by ensuring two conditions:

1. The edges(joins) between nodes of level i and level $i+1$ must be present before the edges (or joins) between nodes of level $i+1$ and level i+2, and so on.

2. The edges (joins) between nodes of level i and level $i+1$ can be present in any order.

(JC1, JC2, JC3, JC4, JC5) and (JC3, JC2, JC1, JC5, JC4) are possible join nesting sequences.

5.7 Assignment of filter predicates to ON/WHERE clause.

Inner Join Predicates:

In the case of an inner join, all predicates pertaining to a table can be considered as part of the join conditions. For example, if I have a filtering rule that only applies to one table then these are all equivalent:

1. SELECT * from part INNER JOIN partsupp ON $p_partkey = ps_partkey$ WHERE $ps_availqty > 9998$;

2. SELECT * from part INNER JOIN partsupp ON $p_partkey = ps_partkey$ AND $ps_availqty > 9998$;

3. SELECT * from part, partsupp WHERE $p_partkey = ps_partkey$ AND $ps_availqty > 9998$;

So, the optimizer will filter results having '*ps_availqty* > 9998' in partsupp first, then join that subset to the part rows based on their mutual partkey column. It would be functionally equivalent to join all of part and partsupp based on partkey first, and then filter that combined result set to only return those that have *ps_availqty* > 9998. Either way, the set of returned results will always be the same.

Outer Join Predicates:

With outer joins we need to be more careful where we put our conditions. Let's compare the first two queries above if they are LEFT OUTER JOINs instead of INNER JOINs.

1. Query 1: SELECT *
   FROM part LEFT OUTER JOIN partsupp
   ON *p_partkey* = *ps_partkey*
   WHERE *ps_availqty* >9998 ;

2. Query 2: SELECT *
   FROM part LEFT OUTER JOIN partsupp
   ON *p_partkey* = *ps_partkey* AND *ps_availqty* > 9998;

Let's discuss these two queries in detail.

- The first query will do the join of all part and partsupp tuples before evaluating the *ps_availqty* > 9998 condition. But, if a row in part does not have a corresponding row in partsupp, that row will still have a corresponding row in the results – until the *ps_availqty* > 9998 condition is applied. At that time, the join result for that row will be excluded because all partsupp values will be NULL due to the outer join. So *ps_availqty* > 9998 will not be true for any Outer joined rows. In this case, the results would be equivalent to those of an INNER JOIN except with the extra resource consumption of performing the OUTER JOIN operation.

- The second query pushes the *ps_availqty* > 9998 condition into the join itself. Thus the only *partsupp* records that will be evaluated as part of the join will be those where $ps_avail qty$ > 9998. Then, any part rows left unpaired will be represented in the join results with $NULL$s for any *partsupp* values.

Query 1 : SELECT * from part LEFT OUTER JOIN partsupp ON p_partkey=ps_partkey WHERE ps_availqty> 9998 ;
Query 2 : SELECT * from part LEFT OUTER JOIN partsupp ON p_partkey=ps_partkey AND ps_availqty> 9998 ;

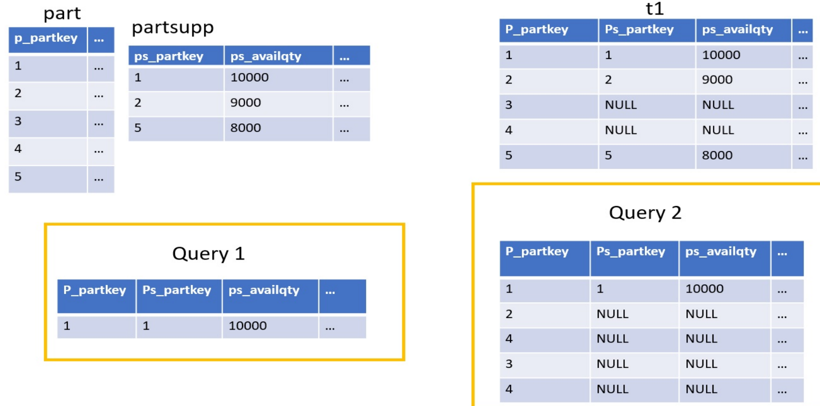


Figure 5.8:

In Figure 5.8, $t1$ represents the results of partial query, "SELECT * FROM part LEFT OUTER JOIN partsupp ON $p_partkey = ps_partkey$".

How to assign filter predicates to ON / WHERE clause

The key observation from the previous discussion is that the filter predicates on columns of tables having low priority in the outer joins must be placed in the ON condition. Also, the filter predicates present in where clause won't allow the column value to be $NULL$, but if the predicate is present in the ON condition, it will allow $NULL$ values. We use this observation to determine the placement of filter predicates. The procedure followed in Algorithm 3 (algorithm to assign filter predicates to ON / WHERE clause):

1. For each filter predicate, set its attribute value to $NULL$.

2. Run Q_H in the mutated D_{1mut}.

3. If the result is empty, the filter predicates in Q_H won't allow the attribute value to be null. Hence it must be placed in the where clause. Else place in ON clause.

The outer join extraction process is shown in Algorithm 4. We can see the refinement of the join type for each edge is done. The FormulateQueries Algorithm is more of syntactic manipulation rather than some logical implementation, so it is not being described in much detail.

5.8 Formulate queries and Eliminate semantically non-equivalent queries (to Q_H):

Syntactic assembly of the query using(Formulate Queries function):

1. Type of joins on each edge.
2. Nesting sequence of joins.
3. Position of filter predicate in ON/ WHERE condition.

To Eliminate queries that are semantically non-equivalent to Q_H. We break the join condition for each edge in the outer join graph in D_1 to produce D_{1mut}. Then, Run all queries formulated above on this mutated D_1 one at a time and store in result R_1 Run Q_H on D_{1mut}, store in result R_2. Eliminate the queries for whom $R_1 != R_2$.

The extracted query Q_E from previous pipeline components including extracted outer joins is given as input to the NEP extraction module. When our running example query is given as input, all the query components will get successfully extracted except <> and *Not Like* operators when we enter the NEP extractor.

5.9 Algorithm.

The provided Algorithm 4 outlines the process for extracting outer joins. Here is a step-by-step explanation:

1. The algorithm starts by initializing the necessary variables, including the global join graph and an empty importance dictionary.
2. It iterates over each edge in the global join graph. For each edge, it extracts the relevant information such as keys and table names.
3. It mutates the join keys of table 1 by negating them. The algorithm then runs the Q_H on the mutated database and stores the result in the $result_H Q$ variable.
4. It analyzes the $result_H Q$. Based on the analysis, the algorithm assigns 'l' or 'h' (low or high) values to temp1 and temp2 variables, indicating the importance of each table in the join.
5. It updates the importance dictionary with the assigned values for Table 1 and Table 2 for the current edge. The new join graph is updated, and the direction of the current edge is set according to the detected join.

6. After iterating through all edges, the algorithm generates a possible edge sequence ($possible_edge_seq$ based on the new join graph and the importance dictionary. It determines the filter predicate placement ($placement_FP$) using the PlaceFilterPredicate function in Algorithm 3.

7. The algorithm formulates possible queries ($set_possible_queries$) based on the possible edge sequence, importance dictionary, and filter predicate placement.

8. It then proceeds to eliminate possible queries that do not produce the same results when comparing the HQ join ($result_HQ$) with the actual query execution ($result_PQ$) for each edge and query combination.

Finally, the algorithm returns the first query from the $set_possible_queries$, which is expected to be the desired outer join query.

Algorithm 3 Filter predicate placement in Outer Joins

$filter_pred_on \leftarrow []$

$filter_pred_where \leftarrow []$

for fp in $global_filter_predicates$ **do**

  D_{1mut} ←Set attributes part of fp to $Null$ in D_1 $new_result \leftarrow Q_H(D_{1mut})$

  if $len(new_result) == 0$ **then**

    $filter_pred_where.append(fp)$

  else

  end

  $filter_pred_on.append(fp)$

  Restore the values of the attributes

end

return $filter_pred_on$, $filter_pred_where$

Algorithm 4 Extraction of Outer Join

```
Global Variables: global_join_graphs
new_join_graph ← global_join_graph
importance_dict ← {}, i ← 0
for edge in range global_join_graphs do
    key1 ← edge[0][0]
    table1 ← edge[0][1]
    key2 ← edge[1][0]
    table2 ← edge[1][1]
    table1.key1 ← −(table1.key1)
    result_HQ ← Run_HQ(D_1)
    p_att_table1_1, p_att_table2 ← Analyze_result(result_HQ)
    importance_dict[edge] = {}
    if len(results_HQ) == 0 then
        temp1 ←' l' and temp2 ←' l'
    else if p_att_table1_1 == Null and p_att_table2 !=Null then
        temp1 ←' l' and temp2 ←' h'
    else if p_att_table1_1 !=Null and p_att_table2 ==Null then
        temp1 ←' h' and temp2 ←' l'
    else if p_att_table1_1 !=Null and p_att_table2 !=Null then
        temp1 ←' h' and temp2 ←' h'
    importance_dict[edge][table1] ← temp1
    importance_dict[edge][table2] ← temp2
    update the new_join_graph, give direction to edge according to join detected.
end
possible_edge_seq ← EdgeSequence(new_join_graph, importance_dict)
placement_FP ← PlaceFilterPredicate(global_filter_predicates)
set_possible_queries ← FormulateQueries(possible_edge_seq,importance_dict,placement_FP)
for edge in range new_join_graph do
    for query in range set_possible_queries do
        table1 = edge[0][0]
        key1 = edge[0][1]
        table1.key1 ← −(table1.key1)
        result_HQ ← Run_HQ(D_1)
        result_PQ ← Run_query(D_1)
        if result_HQ !=result_PQ then
            Eliminate query from set_possible_queries
        end
    end
end
return set_possible_queries[0]
```

Chapter 6

Not Equal Predicates and Correctness

6.1 Not Equal Predicates.

The Not Equal Predicate Extractor has two major components: the NEP database minimizes and the NEP predicate extractor module. The hidden query Q_H, extracted query Q_E, and initial database D_I are given as the input to the NEP Database Minimizer module. This module will find a reduced database D_1 from D_I such that Q_E gives a populated result and Q_H gives an empty result on D_I.

One limitation in [4] which states that " There can be at most one NEP present per attribute. Extraction of the NOT IN operator is out of our extractable domain", was eliminated. The extension of NEP to extract NOT IN operator was performed. Unmasque2 can correctly extract "NOT IN" if present in the hidden query.The reduced database instance D_1 is given as input to the NEP Extractor module. The NEP Extractor module will extract one NEP at a time using database mutation techniques[4].

Using this updated extracted query Q_E again, the presence of some other NEP is checked. This cycle will repeat until all the NEP gets extracted from Q_H.

6.2 Correctness.

Lemma 6.1 *For a hidden query Q_H in EQC_{U2} algebraic predicates will be extracted correctly even if Q_H has disjunction predicates and Not equal predicates.*

Proof: algebraic predicates and disjunction pred/NEP predicates won't have attributes in common according to our extractable query class.

1. Let's say disjunction predicates will affect the extraction of algebraic predicates. In other

words, any filter predicate present in disjunction will lead to the extraction of false algebraic predicates.

2. Filter is a conjunction of disjunctions. filter predicates present in disjunction are of type column op value. Further, for numeric columns, $op \in \{=, <=, <, >, >=, between\}$, whereas for textual columns, $op \in \{=, like\}$.

3. According to extraction process on algebraic predicates in section 4.3.1 The filter predicate extractor is used to get the bounds on the columns. We will first validate whether the bound is concrete or variable. At this stage itself, the bounds will be determined to be variable. Hence the disjunction filter predicate won't be considered ahead in the algebraic predicate extraction. Thus the statement is true that disjunction predicates won't falsely contribute towards the extraction of false/ non-existant algebraic predicates.

4. For NEP predicates, lets assume, The NEP predicate will falsely result into an algebraic predicate

5. NEP predicates are of type column op val. The algebraic predicate extractor in section 4.3.1 will eliminate the attributes present in NEP predicates after the first mutation of the database. It wont further consider those attributes for extraction of algebraic predicates.This assumption is also false.

Therefore, the proof by contradiction establishes that for a hidden query Q_H in EQC_{U2} algebraic predicates will be extracted correctly even if Q_H has disjunction predicates and Not equal predicates.

Lemma 6.2 *For a hidden query Q_H in EQC_{U2} disjunction predicates will be extracted correctly even if Q_H has algebraic predicates and Not equal predicates.*

Proof: The correctness of the above statement can be established using proof by contradiction

1. Assume that if Q_H has algebraic predicates and NEP predicates, the extraction of disjunction predicates will be affected.

2. In the case of algebraic predicates, all attributes participating in algebraic predicates will be eliminated from the check for disjunction predicates. This means that the disjunction extraction process will not consider these attributes. However, this assumption contradicts the fact that the extraction of disjunction predicates will be affected. Therefore, this assumption is false.

3. In the case of NEP predicates, the disjunction will contain filter predicates of the form "column op value,". for numeric columns, $op \in \{=, <=, <, >, >=, between\}$, whereas for textual columns, $op \in \{=, like\}$. According to [Mukul's thesis], filter predicates of this type will be extracted correctly despite the existence of NEP predicates. The disjunction extraction process performs multiple cycles of database mutation, minimization, and filter predicate extraction, ensuring the correct extraction of filter predicates. Therefore, disjunctions will be extracted correctly in spite of NEP predicates.

4. Since both the assumptions regarding algebraic predicates and NEP predicates have been proven false, the original assumption that the extraction of disjunction predicates will be affected is false as well.

Thus, Lemma 6.2 holds true. For a hidden query Q_H in EQCU 2, disjunction predicates will be extracted correctly even if Q_H has algebraic predicates and NEP predicates.

Lemma 6.3 *For a hidden query Q_H in EQC_{U2} Not Equal predicates will be extracted correctly even if Q_H has algebraic predicates and disjunction predicates.*

1. Assume that if Q_H has algebraic predicates and disjunction predicates, the extraction of NE predicates will be affected.

2. In the case of algebraic predicates, all attributes participating in algebraic predicates will be eliminated from the check for NE predicates. This means that the NE predicate extraction process will not consider these attributes. However, this assumption contradicts the fact that the extraction of NE predicates will be affected. Therefore, this assumption is false.

3. In the case of disjunction predicates, the attributes are disjoint to that of NE predicates. The extraction algorithm can correctly identify and extract these NE predicates even if they coexist with disjunction predicates. The presence of disjunction predicates does not hinder the accurate extraction of NE predicates.

4. Since both the assumptions regarding algebraic predicates and disjunction predicates have been proven false, the original assumption that the extraction of NE predicates will be affected is false as well.

5. Thus, Not Equal (NE) predicates will be extracted correctly for a hidden query Q_H in EQC_{U2}, even if Q_H has algebraic predicates and disjunction predicates.

Therefore, the proof by contradiction establishes that NE predicates will be extracted correctly in the presence of algebraic predicates and disjunction predicates for a hidden query Q_H in EQC_{U2}.

Chapter 7

Experiments

7.1 U1 v/s U1.5 Performance Enhancement Results

Correlated sampling increases the probability of having a successful sample. On the sampled database, a view-based recursive halving algorithm works faster than the copy-based halver. Thus we can see the significant extraction time reduction in Unmasque 1.5 when compared to Unmasque 1 in Figure 7.1. Figure 7.1 does not include the result comparison time. The queries in Figures are TPC-H [9] benchmark Queries.

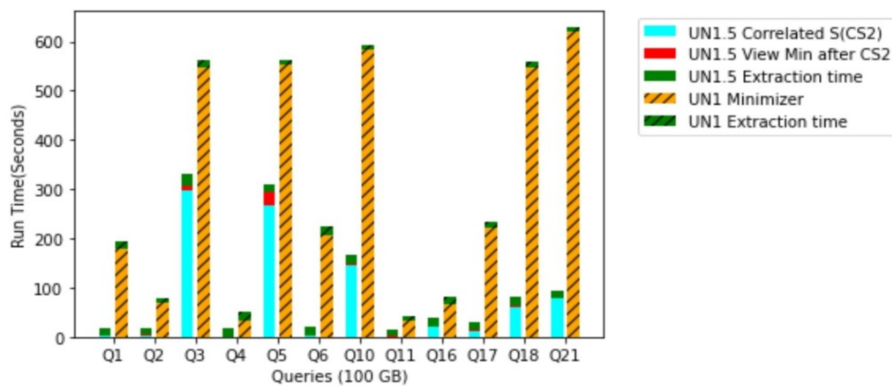


Figure 7.1: Extraction time Unmasque 1 v/s Unmasque 1.5

Figure 7.2 presents the extraction time and hash-based result comparison time for TPCH queries conducted on a 100 GB database.

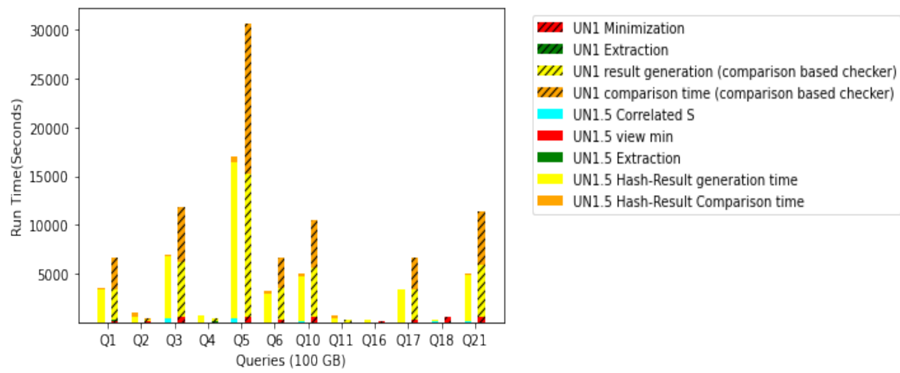


Figure 7.2: Result comparator time

7.2 Unmasque 2

The extraction time of Queries in Unmasque 2 is shown in Figure 7.3. First query in the graph is Unmasque 2's exemplar query.

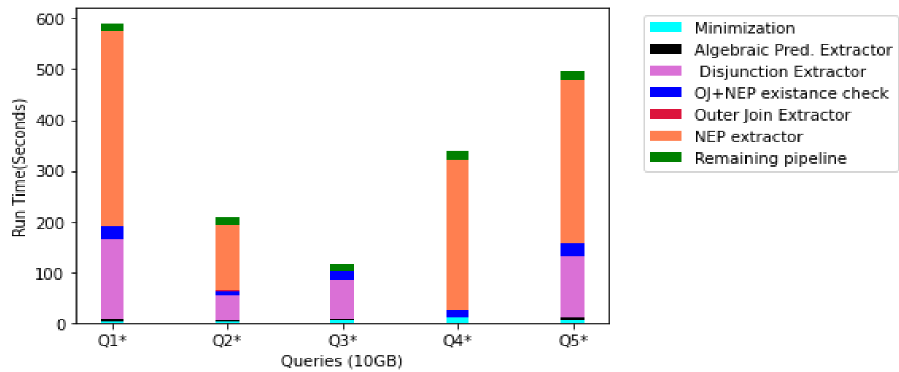


Figure 7.3: Unmasque 2: Extraction time.

Chapter 8

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 D_I and a populated result R_I, identify a candidate SQL query Q_C such that $Q_C(D_I)= R_I$. 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 [10], REGAL [11], and SCYTHE [12]. The ground-truth query is not available in QRE, due to which the output query Q_C is organically dependent on the specific (D_I , 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 (D_I, R_I) instance. Unmasque is a tool to address the HQE problem.

Our work is enhancing the extraction scope of Unmasque and implementing performance optimization techniques.

Chapter 9

Conclusion and Future Work

We have implemented several performance enhancements in Unmasque, such as correlated sampling, a view minimizer, and a hash-based result comparator. Following that, we expanded the extraction capabilities of Unmasque by incorporating algebraic predicates, disjunctions, outer joins, and not-equal predicate extractors. While some of these features were built from scratch, others were updated and integrated into Unmasque 2.

While Unmasque has undergone significant expansion in scope, it still lacks the capability to handle nested queries. The development and incorporation of set operators, in Unmasque 2. One potential area for future development would be to focus on addressing these areas.

Bibliography

[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.

[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 2

[3] Generating Test Data for Killing SQL Mutants: A Constraint-based Approach. http://www.cse.iitb.ac.in/infolab/xdata/

[4] Mukul Sharma: Efficient Extraction of Hidden Negation Predicates https://dsl.cds.iisc.ac.in/publications/thesis/mukul.pdf

[5] Aman Sachan: Extracting Hidden Algebraic Predicateshttps://dsl.cds.iisc.ac.in/publications/thesis/aman.pdf

[6] Sumang Garg: Incorporating Disjunction and Union in Hidden Query Extraction https://dsl.cds.iisc.ac.in/publications/thesis/sumang.pdf

[7] Achyuta Krishna : Non-invasive Extraction of Hidden Queries

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

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

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

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

[12] C. Wang, A.Cheung, and R. Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In Proc. of PLDI Conf. 2

Appendix

Q1*:

SELECT $l_suppkey, l_returnflag, p_partkey, l_quantity, ps_availqty, sum(p_size)$
FROM lineitem
INNER JOIN partsupp ON $l_suppkey = ps_suppkey$
LEFT OUTER JOIN part ON $p_partkey = ps_partkey$ and ($p_size > 49$ OR $ps_availqty > 9900$)
WHERE $l_shipmode$ IN $('MAIL','SHIP','TRUCK')$ AND $l_quantity <> 36$ AND $(l_quantity >= 30)$ AND $l_commitdate <= l_receiptdate$ AND $l_returnflag$ NOT IN ('N')
GROUP BY $l_suppkey, l_returnflag, p_partkey, l_quantity, ps_availqty$
ORDER BY $ps_availqty$
LIMIT 100;

Q2*:

SELECT *
FROM supplier
LEFT OUTER JOIN nation ON $s_nationkey = n_nationkey$ AND $(s_acctbal <= 2000$ OR $n_regionkey = 3)$ AND $n_name <>' RUSSIA'$ AND $s_suppkey > 25$;

Q3*:

SELECT $p_partkey, s_acctbal, ps_suppkey$
FROM part **INNER JOIN** partsupp ON $p_partkey = ps_partkey$ AND $p_size > 7$
LEFT OUTER JOIN supplier ON $ps_suppkey = s_suppkey$ AND $s_acctbal < 2000$

Q4*:

SELECT $l_orderkey, l_linenumber$
FROM lineitem, partsupp WHERE $ps_partkey = l_partkey$ AND $ps_suppkey = l_suppkey$ AND $l_linenumber <> 1$;

Q5*:

SELECT $c_acctbal, o_orderkey, c_name,$
$o_shippriority, c_nationkey, o_orderstatus$

FROM orders
RIGHT OUTER JOIN customer ON $c_custkey = o_custkey$ AND $o_orderstatus <>' O'$
WHERE $c_acctbal < 1000$ AND $c_nationkey < 10$;