Incorporating Disjunction and Union in Hidden Query Extraction

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

BY

Sumang Garg

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

July, 2021

Declaration of Originality

I, **Sumang Garg**, with SR No. **04-04-00-10-42-19-1-16975** hereby declare that the material presented in the thesis titled

Incorporating Disjunction and Union in Hidden Query Extraction

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2019-21**.
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: 10-07-2021 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 Friends & Family;

For their unconditional love.

Acknowledgements

I would like to express my sincere gratitude to my project advisor, Prof. Jayant R. Haritsa for giving me an opportunity to work on this project with him. I am indebted to him not only for the support and guidance he provided, but also for constantly motivating me.

I would also like to thank the Department of Computer Science and Automation and Indian Institute of Science for providing all the necessary facilities and environment even in these tough times.

I am extremely thankful to Kapil Khurana for helping and assisting me through out the project. His valuable suggestions have played a big role in the completion of this project. I would also like to thank all my lab mates for their constant support.

Finally, I would like to thank my family for always supporting and encouraging me.

Abstract

UNMASQUE is a non-invasive extraction algorithm which extracts SQL queries hidden within database applications. Lots of database applications have queries in the form of stored procedures or imperative functions which are then encrypted, making it very hard to know the exact query. Hidden Query Extraction problem aims at extracting those queries exactly. Earlier work on UNMASQUE showed how to extract a wide range of queries under certain assumptions in a platform independent way.
Current version of UNMASQUE is not able to handle SQL constructs such as correlated Nested Queries, Disjunctions, and Unions. This project adds to UNMASQUE the functionality for handling Disjunction and Union operator under certain assumptions.

Contents

List of Figures

Chapter 1

Introduction

Queries in database applications often appear in stored procedures or imperative functions, which in turn are encrypted, making it hard or even impossible to see the query. We can still run the encrypted query and get the output. Such queries are called *Hidden Queries* and the functions in which they are embedded are termed as *executable* for the queries. Hidden Query Extraction(HQE) was recently introduced and is the problem dealing with extracting these hidden queries.

Formally defined, HQE is: *Given a black-box application $\mathcal{A}$ containing a hidden query Q(in either SQL format or its imperative equivalent),and a database instance D on which $\mathcal{A}$ produces a populated result $\mathcal{R}$, unmask Q to reveal the original query (in SQL format).* That is, in contrast to the speculative nature of standard QRE, we intend to find the precise Q such that $\forall_i\ Q(D_i) = R_i$.

Hidden Query Extraction finds a variety of use cases such as: Imperative Code to SQL Translation, Debugging Application with stored SQL procedures, Enhancing Database Security etc..HQE differs from the general Query Reverse Engineering problem because of availability of a hidden ground truth. HQE can be particularly challenging due to challenges such as dependencies between various clauses of the hidden query; as we will see later in the report, a single clause extracted incorrectly can cause the entire extraction to fail.

1.1 UNMASQUE

UNMASQUE(Unified Non-invasive MAchine for Sql QUery Extraction) is a platform independent hidden query extractor introduced in [1]. It uses a judicious combination of database mutation and database generation to extract hidden queries.

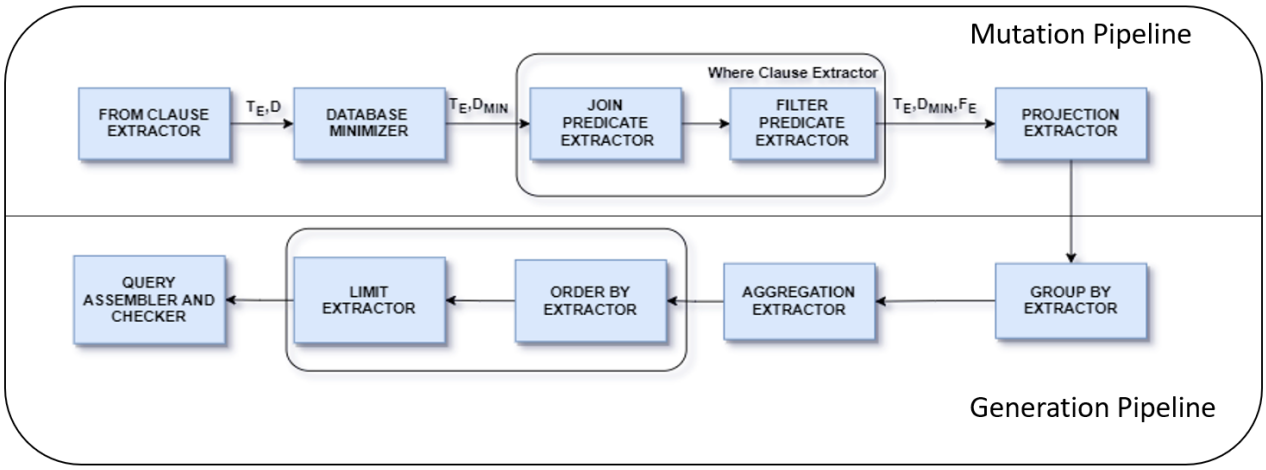

Figure 1.1: UNMASQUE Pipeline

Currently UNMASQUE is able to extract a substantial class of SPJGAOL (Select, Project, Join, Group By, Aggregate, Order By, Limit) queries, under certain assumptions. The UNMASQUE architecture is shown in Figure 1.1.

It has two sequential pipelines for extracting hidden queries: a Mutation pipeline and a Generation pipeline. Mutation pipeline is based on mutations of the original/reduced database and is responsible for handling the SPJ features of the query. The modules in this segment require targeted changes to a specific table or column while keeping the rest of the database intact.

In contrast, the Generation pipeline is based on the generation of carefully-crafted synthetic databases. It caters to the GAOL query clauses. The modules in this segment require generation of new data for all the query related tables under various row-cardinality and column-value constraints. The synthetic-database is crafted in such a manner that it complies with all the conditions discovered in the previous pipeline; additionally data is generated by each module in different manners such that query output on the data can be used to infer something about the module.

There are certain clauses that UNMASQUE can not handle currently, for example Union, Disjunction, and correlated Nested Queries. When presented with queries having such constructs UNMASQUE will either throw an error and declare the query to be out of extraction scope or extract some wrong query. Consider for example, query from Figure 1.2, $\mathbf{Q}_e$. UNMASQUE will not be able to extract $\mathbf{Q}_e$ because it has Disjunction and Union, and hence it falls outside of the class of extractable queries.

```
(select n_name, sum(l_extendedprice * (1 – l_discount)) as revenue
from customer, orders, lineitem, supplier, nation, region
where c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey and
c_nationkey = s_nationkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and
r_name = 'MIDDLE EAST' and o_orderdate ≥ date '1994-01-01' and o_orderdate < date '1994-01-
01' + interval '1' year
group by n_name
order by revenue desc
limit 100)
Union
( select n_name, SUM(s_acctbal)
from supplier, partsupp, nation
where ps_suppkey = s_suppkey and s_nationkey = n_nationkey and (n_name='ARGENTINA' or
n_regionkey = 3 ) and (s_acctbal > 2000 or ps_supplycost < 500)
group by n_name
)
```

Figure 1.2: Example Query: $\mathbf{Q}_e = \mathbf{Q}_u \cup \mathbf{Q}_l$

The class of queries that UNMASQUE can handle is defined in [1] as *Extractable Query Class*(EQC). EQC assumes that : (i) Filter predicates feature only non-key columns and are of the type column *op* value. Further, for numeric columns, $op \in \{=, \leq, \geq, <, >, between\}$ and for textual columns, $op \in \{=, like\}$; (ii) The join graph is a sub-graph of the schema graph (comprised of all valid PK-FK and FK-FK edges); (iii) All ordering columns appear in the projections; (iv) The limit value is at least 3 (v) All joins are key-based equi-joins.
This project takes under consideration a sub-class of EQC, EQC^{-H}(EQC without HAVING clause) and tries to expand its domain by including the ability to handle constructs such as Union and Disjunction.
When presented with $\mathbf{Q}_e$, in the worst case UNMASQUE will extract nothing and declare that the query is out of the extractable domain by raising an error. In practice, UNMASQUE may not be able to identify that query is out of EQC and end up extracting a wrong query. The behaviour of UNMASQUE for $\mathbf{Q}_e$ is speculative and not deterministic because it is a function of dbMinimizer output. The worst case refers to the minimization which will eventually force UNMASQUE to raise an error.

One of the key modules in UNMASQUE is Database Minimizer. It addresses the row-minimality problem which is defined as: *Given a database instance D and an executable Q producing a populated result on D, derive a reduced database instance D_{min} from D such that removing any row of any table present in Q results in an empty output.*
Minimizer module of UNMASQUE takes initial database D(and Q) and finds D^1(assuming

there is not HAVING clause), a minimized database such that all table present in the FROM clause have exactly one row. For the rest of the report D_{min} or the minimized database refers to D^1.

1.2 Related Work

SQUARES[2] is an enumeration based programming-by-example system developed on top of Trinity. It takes desired input and output tables as input and generates corresponding query.

SQUARES is able to formulate even complicated nested queries and thus we considered examining it to see whether the approach can be used to extract nested queries. Upon further experimentation we found that their approach does not scale well and takes an enormous amount of time even for small database sizes.

[3] proposes a solution to reverse engineer complex join queries with arbitrary join graphs. The follow up works ([4], [5]) also handles aggregation.

There is a lot of work done along the lines of reverse engineering SQL queries, but there the problem is fundamentally different from HQE. RQE does not have a ground truth and thus produces one of many possible queries which can produce the given output on given input. The result of QRE depends on the provided input table whereas HQE is completely independent of that(as long as it meets the assumptions).

1.3 Contributions

In this work, we have added to UNMASQUE the ability to extract two more constructs with some assumptions. The proposed modifications are as follows:

- **Disjunction Extraction**: We slightly modify Where Clause Extraction module which allows UNMASQUE to perform well even in the presence of Disjunctions in the query. We propose to add Disjunction Extractor module to UNMASQUE, which extracts Disjunctions in the query (under certain assumption on the database and the query). Disjunction Extractor uses a combination of selection query and dbMinimizer calls to extract Disjunction predicates in a depth-first manner.

- **Union Extraction**: We propose to add a Union Extractor module to UNMASQUE which detects and extracts Union queries. Union Extractor systematically nullifies/restores the table used in the query to detect as well as extract Union queries.

- **Implementation**: The proposed ideas were implemented in Python and added to current UNMASQUE pipeline. They were also tested against various queries to verify correctness and investigate the overheads incurred for added extraction.

1.4 Organization

In chapter 2 we discuss the working of current Filter Clause Extractor and analyze the difficulties behind extracting Disjunctions in the current UNMASQUE framework. This is motivated by an example which shows how the current UNMASQUE will behave in presence of Disjunctions. This is followed the proposed modification to Filter Extraction module which prevents the Disjunction worst case. Then, the algorithm to extract Disjunction along with its assumption in discussed. In the end of the chapter we argue for the correctness of the proposed algorithm.

Chapter 3 begins with a summary of how the From Clause Extractor works and how it can be used to detect set operations. We then discuss how set of tables identified by From Clause Extraction can be used to identify the presence of a Union operator. Once, Union is identified we propose *Tables_Detection* algorithm, which is used to find out the set of tables present in both queries and finally *Union_Extractor* algorithm is discussed, which uses this knowledge to extract the exact queries.

Finally, chapter 4 discusses the design and the results of the empirical evaluation.

Chapter 2

Disjunction

2.1 Background

To show why disjunctions are hard to extract in the current pipeline, first, let's take a look at how the filter predicate extraction works presently. Filter Extractor module runs after the database is minimized. It checks every attribute in all the tables used in FROM clause(Filter extraction is done after FROM clause extraction, so it knows which tables are used in the query) for filters. For each attribute, the extractor checks for presence of a filter by setting its value once to the maximum and once to minimum value of the domain and running the query. If there is a filter on the attribute, the output will be empty for at least one of the two cases(as long as there is no disjunction). Once, a filter is detected on some attribute, extractor finds the filter by binary searching the attribute domain around the true value(value the attribute has in minimized database). In case there is no filter or after the filter is extracted, the attribute value is restored.

This becomes tricky in the presence of Disjunctions. In Figure 2.1.a, we have an example database D, on which lower query from Figure 1.2, $\mathbf{Q}_l$ is run. If the database minimizer leaves us with any one of the four minimizations 2.1.b, 2.1.c, 2.1.d, and 2.1.e then only one predicate from each clause can be extracted. We can see here itself that as the UNMASQUE pipeline is linear and Where Clause Extractor is never called again, the output, if everything else goes smoothly will miss the predicates in Disjunction.
Assume after minimization, we have 2.1.b as the minimized database D'(Note that running the query on 2.1.b gives a populated result, so 2.1.b is a candidate for a minimized database).

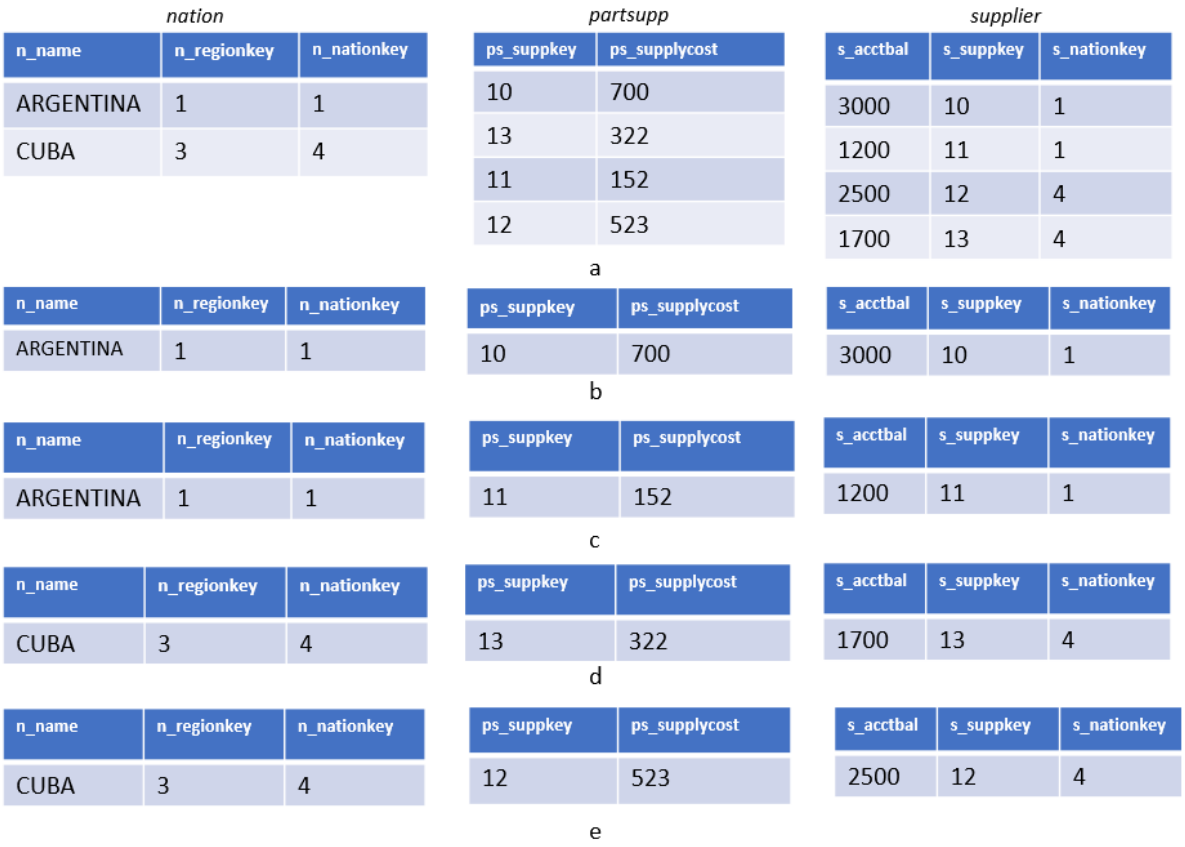

Figure 2.1: Example Database and Minimizations

Now, when the filter extractor changes the value of *n_name* it finds that output disappears and finally concludes that there is a filter on *n_name*. The extractor then goes on to find the filter, *n_name* ='*ARGENTINA*'. But, when it comes to the attribute *n_regionkey*, changing its value will have no impact on the output as the value of *n_name* is already enough to satisfy the clause (*n_name* ='*ARGENTINA*' *or* *n_regionkey* = 3), so the filter extractor will conclude that *n_regionkey* does not have any filter. The same goes for attributes *s_acctbal* and *ps_supplycost*, and thus after the filter extraction is done we will have (*n_nation* ='*ARGENTINA*' *and* *s_acctbal* > 2000) as the filter predicate. Moreover, even if the filter extraction module were supplied the information that *n_regionkey* has a filter on it, it still would not have been able to identify the predicate, as the extractor should know at least one value for which the attribute satisfies the predicate in order to identify it. So, to first identify and then extract the predicates which are in disjunction with the predicates already identified, we need to slightly modify current Filter Extractor and then add a new module.

Disjunction Extractor relies on the output of filter extraction module. It needs one predicate

to be extracted from each clause to extract the entire clause. But, there is an issue with current filter extractor in the presence of disjunctions in query:

Consider, a variation of original database from Figure 2.1, where the **nation** table has a row with n_name ='*ARGENTINA*' and $n\_regionkey = 3$. If the minimized version of this variation of database contains just the row we just added, then the entire first clause will be missed by the extractor.(as filter extraction modifies only one attribute at a time and checks whether the output is null), and for the Disjunction Extractor to extract disjunctions, at least one predicate is required from every clause.

2.2 Modifications to Filter Extraction

To counter this problem, the process for filter predicate extraction has to be changed. If the attribute does not have a non-nullity condition, then after the filter predicate check, instead of restoring the previous value, attributes are instead set to null.
Now, the filter extractor will not find any condition on n_name and will set it to null, but when n_name is set to null, changing $n\_regionkey$'s value will let the extractor know that there is a filter on $n\_regionkey$.
Landing in such minimization is pretty rare and it is the worst case scenario for current UNMASQUE as the Data Generation pipeline produces data in adherence with the filters. Data Generation pipeline is not likely to work if there is a clause from which none of the predicates were identified.

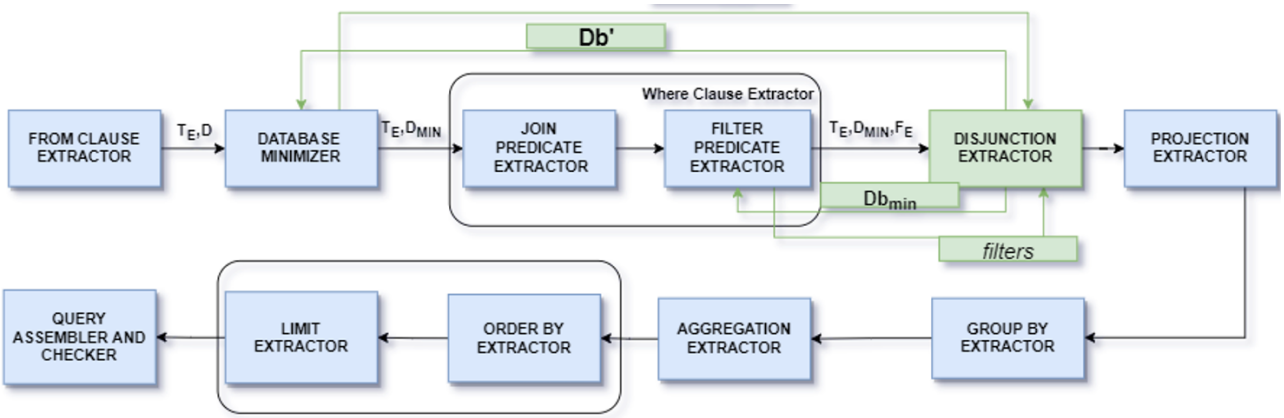

Figure 2.2: Updated UNMASQUE Pipeline

2.3 Disjunction Extraction

Disjunction Extractor is a new module added to UNMASQUE to extract Disjunctions. Addition of Disjunction Extractor changes UNMASQUE pipeline. Earlier the pipeline was linear, whereas now Disjunction Extractor repeatedly calls minimizer and filter extraction modules. The updated pipeline is shown in Figure 2.2. To extract a predicate, we need at least one row which satisfies that predicate and affects the final output. The way filter extractor is defined, it starts with one value that satisfies the filter and then finds the upper and lower limits. So, the predicates in disjunction should also satisfy this condition, i.e. for every predicate, there should be a row that satisfies the said predicate, and removing that row affects the output. But even this will not allow us to extract the complete condition. For example, consider yet another variation of our example database of Figure 2.1 with only such rows that our example query can only have two minimizations(2.1.b and 2.1.d). If we land in the first minimization we extract (n_name ='*ARGENTINA*' *and* $s\_acctbal > 2000$) as the filter and if we land in the second minimization we get ($n\_regionkey = 3$ *and* $ps\_supplycost < 500$) as the filter. There is no simple way for us to know whether the complete filter is ((n_name ='*ARGENTINA*' or $n\_regionkey = 3$) and ($s\_acctbal >$ 2000 or $ps\_supplycost < 500$)) OR ((n_name ='*ARGENTINA*' or $ps\_supplycost < 500$) and ($s\_acctbal > 2000$ or $n\_regionkey = 3$)).

2.3.1 Assumptions

Once it is identified that there is a filter on a particular attribute, the filter extraction module searches for the upper and lower limits of the predicate. The search requires at least one value at which the attribute satisfies the predicate. In our example(2.1.b as minimized database), the filter extraction module will search on right and left of 3000(value of $s\_acctbal$) to get upper and lower limits. So, 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 the two assumptions that the Disjunction Extraction module makes are:

- Filter is a conjunction of disjunctions.
- 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 first assumption is a crucial one from a computational point of view. Disjunction extractor checks for the presence of a disjunction by negating one filter predicate and finding

whether there is some other predicate in disjunction with it. If the filter were not a conjunction of disjunctions, then we will have to negate all possible combination of predicates and then search for disjunctions, which is computationally infeasible.

The second assumption makes sure that all predicates are extract-able. Continuing with our example from Figure 2.1, say the filter extractor has extracted predicates (n_name ='$ARGENTINA$' and $s\_acctbal > 2000$). For the disjunction extractor to be sure that there is a disjunction on n_name, first we have to remove all rows satisfying n_name ='$ARGENTINA$' from the original database(D) and then rerun the query. The rerun will result in a populated output and we conclude that there is a disjunction on n_name. But, if we were to minimize the database now, we may end up with minimization 2.1.d and get filter predicate ($n\_regionkey = 3$ and $ps\_supplycost < 500$), in which case we will have to add additional steps to figure out the exact filter.
So, on top of removing rows that satisfy n_name ='$ARGENTINA$' we also add an additional constraint on the database that we just keep the rows satisfying $s\_acctbal > 2000$. In this case, we are sure to end up with minimization 2.1.e, which will give us filter ($n\_regionkey = 3$ and $s\_acctbal > 2000$), where we can easily see that $n\_regionkey = 3$ is in disjunction with n_name ='$ARGENTINA$'. The second assumption makes sure that all these minimizations are possible. In a large database, the assumptions generally hold.

2.3.2 Extraction

Assuming that the two assumptions hold, the algorithm to extract disjunction is given in Algorithm 1. Disjunction Extractor is called after Filter Extractor Module and thus receives as input F_E, which necessarily contains exactly one predicate from each clause because of the modifications we made to Filter extraction. Disjunction Extractor finally outputs a two dimensional list of all the filters and a string with complete filter.
The algorithm makes multiple calls to dbMinimizer and Filter Extraction modules, but as shown in the pipeline in 2.2 the calls are both subroutine calls and they transfer the control back to the algorithm i.e. Minimizer returns the output to Disjunction Extractor instead of passing it on to Join Predicate Extractor.

Algorithm 1: $Disjunction\_Extraction(FilterList[], D)$

$i = 0$
$Disjunction = [][]$
$n = len(FilterList)$
$final\_filter =' True'$
while $i \leq n - 1$ **do**
    $outer\_con = true$
    $outer\_con = \bigwedge_{0 \leq k < n, k \neq i} FilterList[k]$
    $Disjunction[i][0] = FilterList[i]$
    $j = 1$
    $final\_filter+ = 'and\ (' + FilterList[i]$
    $Db = select\ from\ D\ where\ outer\_con$
    while *true* **do**
        $inner\_con = true;$
        $inner\_con = \neg \bigvee_{0 \leq k < j} Disjunction[i][k]$
        $Db' = select\ from\ Db\ where\ inner\_con$
        if $exec(Db') \neq \phi$ **then**
            $Db_{min} = dbMinimizer(Db')$
            $filters = FilterClauseExtractor(Db_{min})$
            $Disjunction[i][j] = filters - FilterList[i]$
            $j++$
            $final\_filter+ = 'or' + Disjunction[i][j]$
        else
            $final\_filter+ = ')'$
            $break$
        end
    end
    $i++$
end

On our running example, consider that the filter extracted by $WhereClauseExtractor$ is ($n\_name =$'$ARGENTINA$' and $s\_acctbal > 2000$), then Algorithm 1 proceeds in the following fashion:

- Delete the rows from database having $n\_name =$'$ARGENTINA$' and keep only the rows

having $s\_acctbal > 2000$.

- Minimize this database and call filter extractor module.
- $FilterClauseExtractor$ returns ($n\_regionkey = 3$ and $s\_acctbal > 2000$) hence $n\_regionkey = 3$ is added to the Disjunction list of $n\_name =$'$ARGENTINA$'.
- Delete the rows from database having $n\_name =$'$ARGENTINA$' or having $n\_regionkey = 3$ and keep only the rows having $s\_acctbal > 2000$.
- Executable output will be empty here and hence this clause is concluded.
- In similar fashion, the second clause will be extracted completely.

So, the selection step in disjunction extraction makes sure that dbMinimizer lands in a particular minimization. In our running example, if we assume that the first minimization we naturally landed in is 2.1.b, disjunction extractor will have three iterations, each time making sure that dbMinimizer lands in a different minimization. The complete filter string is stored in $final\_filter$.

2.4 Correctness

We first note that Algorithm 1 first checks for disjunction by removing all the rows satisfying a particular predicate. If there is no disjunction in the query, then Algorithm 1 only puts a checking overhead on the extraction procedure and does not affect the working of UNMASQUE in any way.
Now, we assume that Algorithm 1 extracts as $final\_filter$: (a_1 or a_2) and (b_1 or b_2).
Firstly, we claim that everything Algorithm 1 extracts is actually a filter. So, lets assume that a_2 is incorrect. There can be two such cases:

- One case would be that there is no a_2 i.e. there is nothing in disjunction with a_1, but the initial check for disjunction(of a_1) must have removed all rows satisfying a_1 and must have retained only the rows satisfying b_1. Algorithm 1 only works when after the selection, query output is populated. So, it must be the case that even after removing all the rows satisfying a_1, the query gives populated output, thus there must be something in disjunction with a_1.

- Second case would be that there is indeed something in disjunction with a_1 but it is not a_2. Algorithm 1 makes sure that only the rows not satisfying a_1 and satisfying b_1 are passed on to minimizer. So, the correctness of a_2 depends on the correctness of filter extraction module. If the filter extraction module is correct, then a_2 must be the predicate.

Last claim we make is that complete filter will be extracted by Algorithm 1. So, we assume that the actual filter was (a_1 or a_2) and (b_1 or b_2 or b_3) and b_3 was just not extracted. Here, we note that after b_2 was extracted by the disjunction extractor, it must have deleted all rows satisfying b_1, b_2, and $\overline{a_1}$. So, there must be rows satisfying (a_1 and b_3) as per our assumption and thus Algorithm 1 will conclude that there is indeed a predicate left to extract, so it will not stop at b_2.

2.5 Optimization

To check disjunction, we load original database and then run our selection query on it(negating certain predicate) and finally check the executable output on this database. To improve the run-time further, instead of loading original database, we load sampled database. Sampling the database is the first step during minimization. Minimizer samples each table and if the samples returns a populated output for executable, those samples are used as starting point of minimimization. These sampled databases are separately stored and used to check for disjunctions. If the check fails, i.e. we find no disjunction, original databases is loaded and checked. As the sampled databases themselves are small, the overhead for checking is not much but we get a speedup in identifying disjunctions in case it is present in sampled database.

Chapter 3

Union

3.1 Background

To understand how we can go about extracting queries with Union, lets first take a look at how FROM clause is extracted in current UNMASQUE. Currently, there are two methods to extract FROM clause in UNMASQUE, which are:

- **Nullify Method**:
  To check whether a base table t is present or not in FROM clause, we nullify the table, i.e. t is set to null and the query executable is run. If, t is present in the query and there are no set operations then the output will necessarily be null. Once the query is run t is restored. The downside of this method is that the query runs to completion every time (for all tables) and thus it takes a lot of time.

- **Rename Method**:
  This is the default method in UNMASQUE to extract FROM clause. To check whether a table t is present or not in the query, we temporarily rename it. Then the query executable is run, if t is part of the query, then the executable will throw an error, which UNMASQUE catches. t is then reverted to its original name. This method will report all the tables used in the query, irrespective of whether there is a set operation or not. Also, to make sure the query does not run to completion for tables not present in query, we use a timeout.

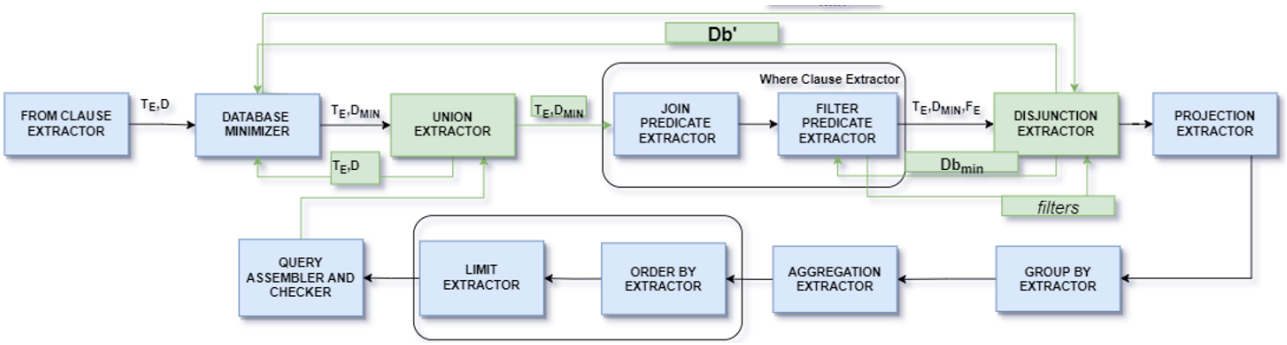

Figure 3.1: UNMASQUE Pipeline with Union

Rename Method and Nullify Method will give different outputs when there is a set operator present in the query, and that is the core idea upon which Union Extractor works.
We represent query with Union as $Q = X \cup Y$, where the order of X and Y does not matter. Further, we represent set of tables present in X(similarly, Y) as T_x(similarly, T_y) and T_{common} refers to set of tables in $T_x \cap T_y$. Now, we can use a combination of these two methods to clearly decide which tables are part of which query.

3.2 Assumptions

Union extractor operates under the assumption that T_x and T_y are not subsets of each other. We also make a slight change to the initial condition of UNMASQUE, where initially we required a database with populated output, now we require an initial database such that both queries of Union produce populated outputs individually.

3.3 Extraction

Union extractor is first invoked after database minimization as can be seen in the updated pipeline in Figure 3.1. So, Union extractor receives as input a minimized version of database and T_E, which is set of all tables used in Q as determined by Rename Method of FROM Clause Extractor. Union extractor then determines whether there is a Union present in the query or not.

To detect whether a Union is present, first we note that if Union operator is indeed present, then the minimized database(the one Union Extractor receives) must fall in one of the following three possible cases:

- Both X and Y produce the same populated output on minimized database.

- X and Y produce different populated outputs on minimized database.
- Only X produces a populated output on the minimized database.

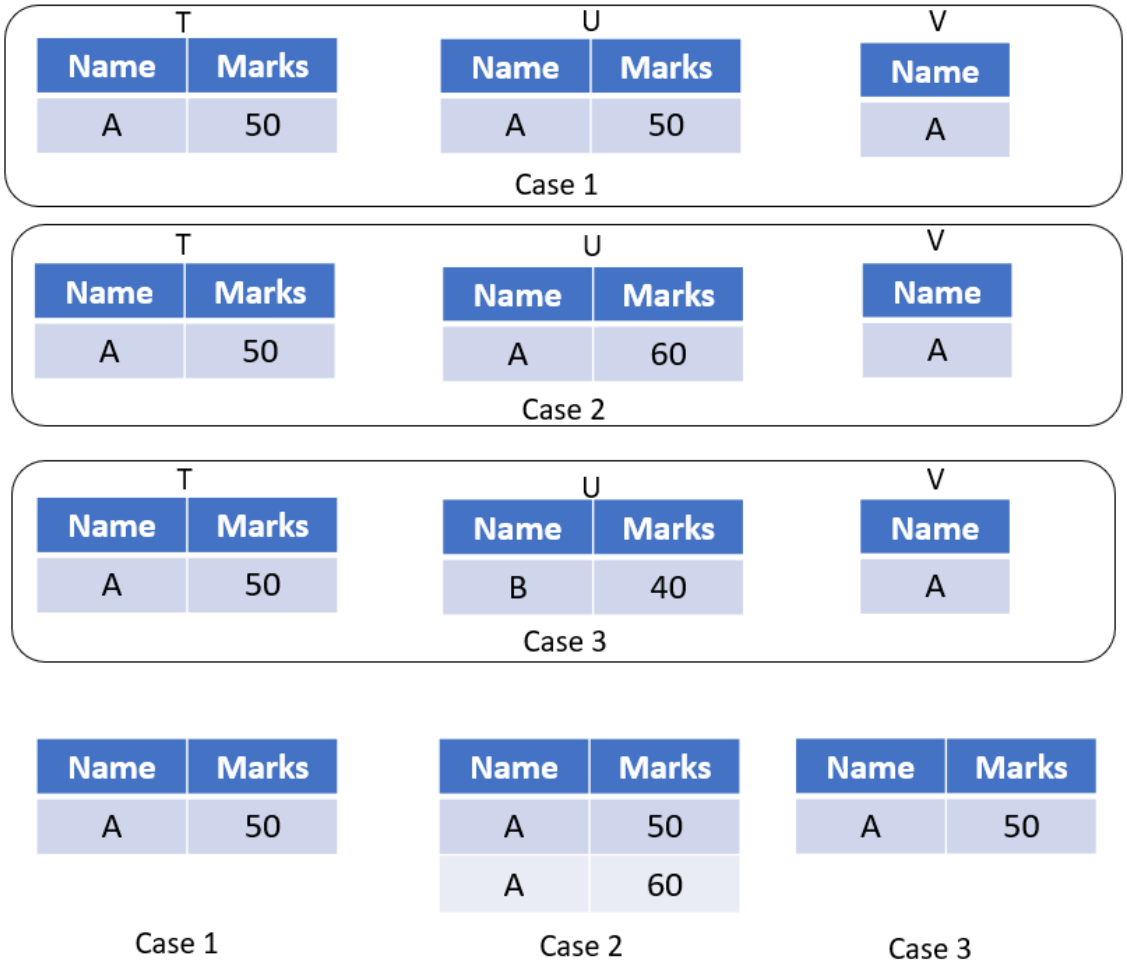

Figure 3.2: Union Example

Figure 3.2 shows an example for all three cases and corresponding outputs when Q is **(Select * from T,V where T.A=V.A) Union (Select * from U,V where U.A = V.A)**. It is easy to see that in all three cases there must exist a table t in T_E, which can be set to null while still maintaining a populated output. More precisely, all the tables present in $T_y - T_x$ can be set to null while still keeping the output populated. We use this observation to detect whether a Union is present in the query or not in *Union_Detection* algorithm.
After *Union_Detection* is run, *single_query_tables* is the set of tables that appear in either X or Y, but not both, whereas, *common_tables* are either the tables appearing in both queries(for Case 1 and 2) or the tables appearing in X(for Case 3). Continuing our initial example from

Figure 1.2, *single_query_tables* will be **{customer, orders, lineitem, partsupp, region }** and *common_tables* will be **{nation, supplier}** for case 1 or 2, but they will correspondingly be **{supplier}** and **{customer, orders, lineitem, partsupp, nation, region }** for case 3(assuming that Q_u is Q_x).
If the current UNMASQUE tries to extract $\mathbf{Q}_e$ from Figure 1.2, it will be able to identify the set of all tables, and in general will end up extracting some query which will have all the tables from T_E in the FROM clause but other constructs will be of only one query(it can be any one of the two) as shown in appendix. But the relatively rare case, when the minimization lands in case 1 or 2, will be problematic. In this case, it may happen that UNMASQUE fails to extract any query at all.

Algorithm 2: $Union\_Detection(T_E, D)$

```
isUnion = false
common_tables = [], i = 0
single_query_tables = []
while i < len(T_E) do
    e = T_E[i++]
    Nullify(e)
    if exec(D) ≠ φ then
        single_query_tables.append(e)
    else
        common_tables.append(e)
    end
    Restore(e)
end
if len(single_query_tables) ≥ 1 then
    isUnion = true
end
```

For example, when the *FilterClauseExtractor* tries to toggle values of attributes in table set of $\mathbf{Q}_u$, the output from $\mathbf{Q}_l$ may prevent final result from becoming empty and thus not allowing UNMASQUE to figure out the filters, without which the extraction is likely to not work at all. Landing in case 1 or 2 is relatively rare because dbMinimizer only cares for a

populated output when minimizing. If during the minimization process, output from one query disappears due to some choice that dbMinimizer makes, it will not matter to dbMinimizer.
If Union is not present, then UNMASQUE goes about its business as usual and Union extractor does nothing more. But, if Union is present, then Union extractor further tries to determine T_x and T_y. To decide T_x and T_y we first determine which case out of the three, the minimized database is in and we call $Tables\_Detection$.
The idea behind $Tables\_Detection$ is that if we are in case 1 or 2, then $single\_query\_tables$ must be $(T_x \setminus Ty) \cup (T_y \setminus Tx)$. Else, if we are in case 3, then $single\_query\_tables$ will only contain $T_y \setminus Tx$.

Algorithm 3: $Tables\_Detection(single\_query\_tables, D)$

```
T1 = [single_query_tables[0]]
T2 = [], isCase3 = true
i = 1
Nullify(single_query_tables[0])
while i < len(single_query_tables) do
    e = single_query_tables[i++]
    Nullify(e)
    if exec(D) ≠ φ then
        T1.append(e)
    else
        T2.append(e)
    end
    Restore(e)
end
if len(T2) ≥ 1 then
    isCase3 = false
end
```

If we are not in case 3, then $T_x = common\_tables \cup T1$ and $T_y = common\_tables \cup T2$, otherwise all we know for sure is that $T_x = common\_tables$ and T_y is union of $T1$ and some subset of $common\_tables$. So, if we are not in case 3, Algorithm 3 finds T_x for our Figure 1.2

example as **{customer, orders, lineitem, supplier, nation, region }** and T_y as **{supplier, partsupp, nation }**. If we are in case 3, then T_x is the same, where as T_y is not known.

Algorithm 4: $Union\_Extractor(T_E, D^1)$

if $isCase3$ **then**
- $control\_stub(isCase3, T_x, 1)$
- *load Database D*
- T_E=$FromClauseExtractor(method{=}Nullify)$
- $T_y = T_E \cup T1$
- $T_x = T_x \setminus T_E$
- $T_E = T_y$
- $Nullify(T_x[0])$
- $D' = dbMinimizer(D)$
- $control\_stub(isCase3, T_y, 0)$

else
- $Nullify(T2[0])$
- $control\_stub(isCase3, T_x, 1)$
- $Restore(T2[0])$
- $Nullify(T1[0])$
- $control\_stub(isCase3, T_y, 0)$

end

Once we find T_x, we extract exact queries with $Union\_Extractor$. If we are not in case 3, then Union extractor's work is almost done. It Nullifies any one table from $T2$ and then calls *control_stub*. *control_stub* calls the rest of the UNMASQUE pipeline(starting with $WhereClause\ Extractor$) with T_E set as its second argument. As, the third argument is 1, *control_stub* returns the control to Union extractor after this run of UNMASQUE. As T_x was set as T_E and there was no impact from Y(because $T2[0]$ is nullifed), the query that UNMASQUE extracts now is guaranteed to be X. Once X is extracted, control is transferred back to Union extractor, which restores *common_tables* to the state they were in after minimization, it sets T_E as T_y and calls the next module. In this second run of UNMASQUE, Y is extracted.

Things go slightly different if we land in case 3. The initial step for extracting X are essentially the same, T_E is set to T_x(which is *common_tables* here), as the tables from Y have

no impact here there is no point in nullifying them and the *control_stub* is called. Once, this run is complete, i.e. X is extracted, it moves on to the next step.
To extract Y, the extractor needs to know T_x and T_y and to do it, all the tables are restored to their initial state i.e. the state before the first minimization. At this stage, due to our assumption, Y has necessarily populated output. Here, the Nullify Method of FROM clause extractor is called and the output is stored in a T_E. Because of the way Nullify Method is defined, tables in T_E list are exactly the tables which appear in both queries.
Hence, T_y now becomes $T_E \cup T1$ and set of tables unique to X is $T_x \setminus T_E$. Extractor then nullifies any one of the tables unique to X, sets T_E to T_y and calls dbMinimizer. As dbMinimizer only minimizes the set of table in T_E, it minimizes all the tables in T_y. Following the same reasoning as earlier, we note that this time the extracted query will be Y.

3.4 Correctness

First claim is that if there is no Union in the query, then *Union_Detection* will not report any Union. This is easy to see because *Union_Detection* only reports Union when there exists some table which when not present in database, makes query execution throw an error, but when nullified does not make the output empty. This behaviour is not possible for a query that does not have have any set operations, and as Union is the only set operation permissible, there must be a Union in the query.
Next claim is that sets of tables are correctly identified by *Tables_Detection*(for case 1 and 2). When both queries contribute to output, then to nullify output, either we nullify any table from $T_x \cap T_y$ or we nullify at least one table each from $T_x \setminus common\_tables$ and $T_y \setminus common\_tables$. As for case 1 and 2, $single\_query\_tables$ is $(T_x \cup T_y) \setminus common\_tables$. Sets are detected by first putting one table in $T1$, then nullifying it, and then noticing which other table when nullified together with this one table, will make output null.
Lets assume that some table e is wrongly added to $T2$ by *Tables_Detection* and it actually belongs in $T1$. To be added in $T2$, nullifying e must have made the output null. As the only other table nullified in this iteration was the first table(which is empty for all iterations), e must be in a different set than the first table, and as first table is in $T1$, e must necessarily be in $T2$ and thus we contradict the assumption. Similar argument can be used to argue that all tables added in $T1$ must necessarily belong there.
So, when there is no Union, Union detector does not affect the working of UNMASQUE at all and when there is a Union it identifies tables and uses the rest of UNMASQUE as a black box.

3.5 Optimization

Similar to the optimization in Disjunction Extractor, in case of second minimization(for case 3), before loading the original database back, first the sampled tables are loaded to see if Y produces populated output on them. Only failing that original database is loaded back.

Chapter 4

Experiments

The new modules were tested against a set of Union and Disjunction queries to verify correctness and to see how much overhead is incurred due to the additions.
All the experiments were run on PostgreSQL[6] hosted on an Intel Xeon 2.3 GHz CPU, 32GB RAM, Ubuntu Linux equipped machine. Experiments were conducted on TPC-H[7] benchmark queries which were slightly tweaked to remove nesting and in some cases Disjunction was explicitly introduced. For Union experiments select clause in TPC-H queries were slightly changed such that Union operator can be applied.
As UNMASQUE extracts the hidden ground truth, it is independent of original database as long as assumptions are met, so to conduct better evaluation we need complexity in queries, which TPC-H provides. Additionally, TPC-H queries test the performance of new modules as part of the UNMASQUE system, rather than just checking the performance of standalone modules.
All of UNMASQUE's original code-base, other than the change in $FilterExtraction$ module, was used as a black-box. The modules for Union and Disjunction extraction, make routine calls to UNMASQUE modules. The algorithms were implemented in Python 3.6 and have been integrated with UNMASQUE code-base.
All the extracted queries were manually verified to be correct. Both the Disjunction and Union queries are listed in appendix. Also listed in appendix is output of one run of original UNMASQUE on Disjunction and Union queries. The experiments were performed on TPC-H database of sizes 1 and 10 GB(SF 1 and SF 10).

4.1 Disjunction

Queries to evaluate Disjunction performance were slight modification of original TPC-H queries. Disjunctions were introduced on various types of attribute to cover all edge cases. As IN operator is also a Disjunction, the module was able to extract it too. The time taken to extract the queries and disjunction is listed in Figure 4.1 (total extraction time= Extraction Time + Disjunction Time). First, we note that the disjunction extraction module itself takes a lot more time than the total time taken by all other modules combined. There is varied range of time consumed(6s-167s) and this is in major part due to selection queries on lineitem table.
Selection queries are necessary to make sure that minimization we end up with is not something that we have seen earlier. But in case when there is no index on the attribute selection query is run on, it is a very time-consuming operation. As lineitem table has the most rows out of all tables in TPC-H, selection operator is especially costly when this table is involved in the query.

Query	SF=1		SF=10	
	Extraction Time(s)	Disjunction Time(s)	Extraction Time(s)	Disjunction Time(s)
1	5.03	11.24	18.54	159.2
2	9.34	10.3	74.3	152.5
3	4.9	5.74	29.3	31.43
4	5.13	14.61	16	139.3
5	2.44	1.12	12.28	5.78
6	2.4	6.35	5.4	16.43

Figure 4.1: Time Consumed

As, we can see from the stated algorithm, Disjunction Extraction has two very costly steps: dbMinimizer and Selection Queries. To further understand the time taken by Disjunction extraction, the time taken for different minimizations and different selections were summed up separately and then compared. The results are plotted in Figure 4.2 and 4.3.

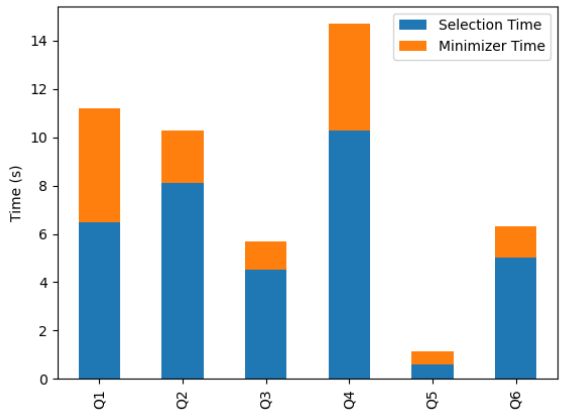

Figure 4.2: SF1 Time Breakup

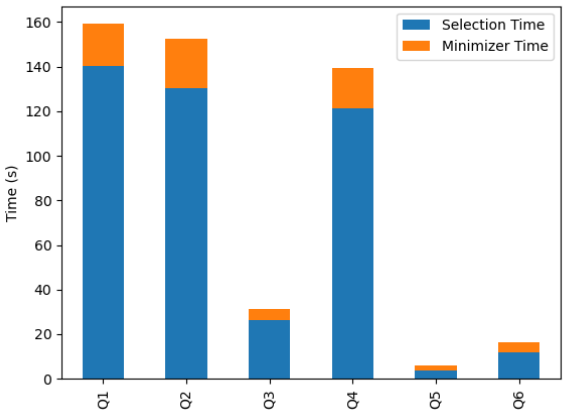

Figure 4.3: SF10 Time Breakup

One thing to note here was that individual database minimizations took far less time than the initial minimization and that is because selection reduced the size of source table and that was because after each selection the table sizes were reduced.

4.2 Union

Union experiments were also done for scale factors 1 and 10. As expected, most of the queries land in case 3. Query U1 has just the select statements and no filters or joins, thus it lands in case 1 or 2 and hence the Union overhead for it was size invariant. Time taken for extraction and the Union overhead for different queries is listed in Figures 4.4 and 4.5.

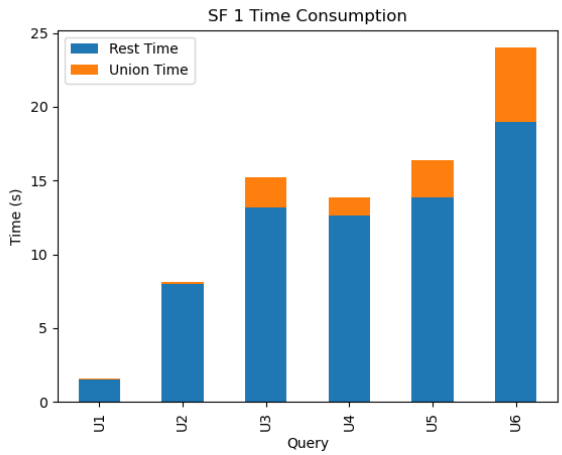

Figure 4.4: Union Overhead(SF 1)

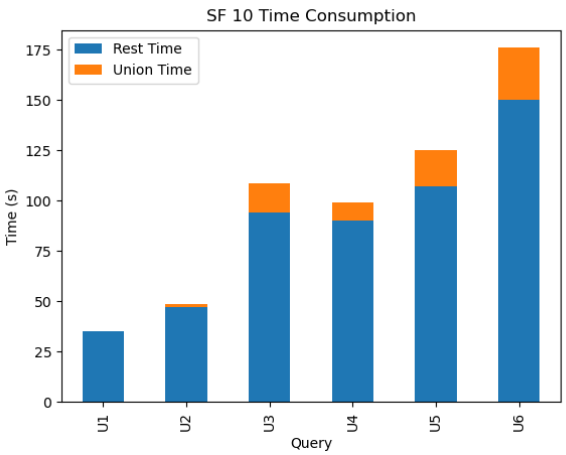

Figure 4.5: Union Overhead(SF 10)

Overall Union overhead depends on the size of database but is still pretty small and practical for offline analysis environment.

4.3 Union vs Union All

Extracted Union queries were not exactly syntactically equivalent to the hidden queries and this because the Union operator removes duplicates.
So, the queries extracted had some attributes in group by clause which were not present in original queries and that accounts for this duplication removal. For example, extracted U1 actually was
(select c_acctbal from customer **group by c_acctbal**) **Union** (select l_extendedprice from lineitem **group by l_quantity**)
Here the portion in bold was not present in original query. But the extracted query is still semantically equivalent to original query. Moreover, these redundant group by's are later removed during canonicalization.
Union All operator is the same as Union operator except that it does not remove duplicates. So, the same queries with Union All operator instead of Union operator were extracted as they were.

Chapter 5

Conclusion and Future Work

UNMASQUE now has the ability to extract Disjunction and Union operator under certain assumptions. There are some restriction on Disjunction operator, but the modifications in Filter extraction makes sure that even when the assumptions are not met, UNMASQUE extracts at least one predicate from each clause, which allows Data Generation pipeline to work and hence some portion of query is still always extracted. Similarly with Union operator, if the assumption is not met, then at least one out of two queries will always be extracted.
There are some operators that can not be extracted by UNMASQUE yet. One possible direction for future work would be to come up with new ideas to extract set operations like set difference and intersection. More fundamentally, formally identifying the capabilities of non-Invasive extraction remains to be solved.

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 1

[2] Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins and Vasco Manquinho. *SQUARES : A SQL Synthesizer Using Query Reverse Engineering*. PVLDB, 13(12): 2853-2856, 2020. 4

[3] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. *Reverse Engineering Complex Join Queries*. In SIGMOD, 2013. 4

[4] C. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava. *Reverse Engineering Aggregation Queries*. PVLDB, 10(11), 2017. 4

[5] W. C. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava. *REGAL+: Reverse Engineering SPJA Queries*. PVLDB, 11(12), 2018. 4

[6] http://www.postgresql.org/ 22

[7] http://www.tpc.org/tpch/ 22

Chapter 6

Appendix

6.1 Disjunction 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_extendedpr as avg_price, avg(l_discount) as avg_disc ,count(*) as count_order
from lineitem
where l_shipdate IN (date '1998-12-01', date '1998-11-11', date '1992-01-06')

group by l_returnflag, l_linestatus
order by l_returnflag,l_linestatus;

Q.2 :
select l_orderkey,sum(l_extendedprice) as revenue,o_orderdate,o_shippriority
from customer,orders,lineitem
where (c_mktsegment = 'FURNITURE' or c_mktsegment = 'AUTOMOBILE') and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < date '1995-03-29' and l_shipdate > date '1995-03-29'
group by l_orderkey,o_orderdate,o_shippriority
order by revenue desc,o_orderdate limit 10;

Q.3:
select l_shipmode,sum(l_extendedprice) as revenue
from lineitem

where l_shipdate $\geq$ date '1994-01-01' and l_shipdate $<$ date '1994-01-01' + interval '1' year and (l_quantity =42 or l_quantity =50 or l_quantity=24)
group by l_shipmode limit 100;

Q.4:
select AVG(l_extendedprice) as avgTOTAL
from lineitem,part
where p_partkey = l_partkey and (p_brand = 'Brand#52' or p_brand = 'Brand#12') and (p_container = 'LG CAN' or p_container = 'LG CASE');

Q.5 :
select c_mktsegment,MAX(c_acctbal)
from customer
where c_nationkey IN (1,5,9,10)
group by c_mktsegment;

Q.6:
select n_name,SUM(s_acctbal)
from supplier,partsupp,nation
where ps_suppkey=s_suppkey and
s_nationkey=n_nationkey and (n_name ='ARGENTINA' or n_regionkey =3) and (s_acctbal $>$ 2000 or ps_supplycost $<$ 500)
group by n_name;

6.2 Union Queries

U1:
(**select** c_acctbal
from customer)
Union
(**select** l_extendedprice
from lineitem)

U2:
(**select** sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price
from lineitem

where l_shipdate $\leq$ date ´1998-12-01´ interval ´71 days´
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus)
Union
(**select** s_acctbal,p_partkey
from part, supplier, partsupp, nation, region
where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_size = 38 and p_type like '%TIN' and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'MIDDLE EAST'
order by s_acctbal desc, n_name, s_name, p_partkey)

U3:
(**select** l_orderkey, $sum(l\_extendedprice \cdot (1 - l\_discount))$ as revenue
from customer, orders, lineitem
where c_mktsegment = 'BUILDING' and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < date '1995-03-15' and l shipdate > date '1995-03-15'
group by l_orderkey, o_orderdate, o_shippriority
order by revenue desc, o_orderdate)
Union
(**select** s_acctbal,p_partkey
from part, supplier, partsupp, nation, region
where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_size = 38 and p_type like '%TIN' and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'MIDDLE EAST'
order by s_acctbal desc, n_name, s_name, p_partkey)

U4:
(**select** c_name,sum(l_extendedprice $*$(1$-$ l_discount)) as revenue
from customer, orders, lineitem, nation
where c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate $\geq$ date '1994-01-01' and o_orderdate < date '1994-01-01' + interval '3' month and l_returnflag = 'R' and c_nationkey=n_nationkey
group by c_name, c_acctbal, c_phone, n_name, c_address, c_comment
order by revenue_desc)
Union

(**select** ps_COMMENT, sum(ps_supplycost * ps_availqty) as value
from partsupp, supplier, nation
where ps_suppkey = s_suppkey and s_nationkey = n_nationkey and n_name = 'ARGENTINA'
group by ps_COMMENT
order by value desc)

U5
(**select** s_acctbal,p_partkey
from part, supplier, partsupp, nation, region
where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_size = 38 and p_type like '%TIN' and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'MIDDLE EAST'
order by s_acctbal desc, n_name, s_name, p_partkey)
Union
(**select** c_acctbal,sum(l_extendedprice ∗(1− l_discount)) as revenue
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
group by c_name, c_acctbal, c_phone, n_name, c_address, c_comment
order by revenue_desc)

U6
(**select** n_name, sum(l_extendedprice * (1 – l_discount)) as revenue
from customer, orders, lineitem, supplier, nation, region
where c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey = s_nationkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'MIDDLE EAST' and o_orderdate *geq* date '1994-01-01' and o_orderdate < date '1994-01-01' + interval '1' year
group by n_name
order by revenue desc
limit 100)
Union
(**select** n_name, SUM(s_acctbal)
from supplier, partsupp, nation

where ps_suppkey = s_suppkey and s_nationkey = n_nationkey and (n_name='ARGENTINA' or n_regionkey = 3) and (s_acctbal > 2000 or ps_supplycost < 500)
group by n_name)

6.3 Disjunction Original UNMASQUE Output

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_extendedpr as avg_price, avg(l_discount) as avg_disc ,count(*) as count_order
from lineitem
where l_shipdate = date '1998-12-01'
group by l_returnflag, l_linestatus
order by l_returnflag,l_linestatus;

Q.2 :
select l_orderkey,sum(l_extendedprice) as revenue,o_orderdate,o_shippriority
from customer,orders,lineitem
where c_mktsegment = 'FURNITURE' and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < date '1995-03-29' and l_shipdate > date '1995-03-29'
group by l_orderkey,o_orderdate,o_shippriority
order by revenue desc,o_orderdate limit 10;

Q.3:
select l_shipmode,sum(l_extendedprice) as revenue
from lineitem
where l_shipdate ≥ date '1994-01-01' and l_shipdate < date '1994-01-01' + interval '1' year and l_quantity =42
group by l_shipmode limit 100;

Q.4:
select AVG(l_extendedprice) as avgTOTAL
from lineitem,part
where p_partkey = l_partkey and p_brand = 'Brand#52' and p_container = 'LG CASE';

Q.5 :
select c_mktsegment,MAX(c_acctbal)
from customer
where c_nationkey = 5
group by c_mktsegment;

Q.6:
select n_name,SUM(s_acctbal)
from supplier,partsupp,nation
where ps_suppkey=s_suppkey and
s_nationkey=n_nationkey and n_name ='ARGENTINA' and s_acctbal > 2000
group by n_name;

6.4 Union Original UNMASQUE Output

U1:
select c_acctbal as l_extendedprice
from customer, lineitem

U2:
select sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price
from lineitem,part,supplier,partsupp,nation,region
where l_shipdate ≤ date ´1998-12-01´ interval ´71 days´
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus

U3: **select** s_acctbal as l_orderkey,p_partkey as revenue
from part, supplier, partsupp, nation, region, customer,orders,lineitem
where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_size = 38 and p_type like '%TIN' and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'MIDDLE EAST'
order by s_acctbal desc, n_name, s_name, p_partkey

U4:
select c_name,sum(l_extendedprice ∗(1− l_discount)) as revenue
from customer, orders, lineitem, nation, partsupp, supplier, 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
group by c_name, c_acctbal, c_phone, n_name, c_address, c_comment
order by revenue_desc

U5

select s_acctbal as c_acctbal,p_partkey as revenue
from part, supplier, partsupp, nation, region, customer, orders, lineitem
where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_size = 38 and p_type like '%TIN' and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'MIDDLE EAST'
order by s_acctbal desc, n_name, s_name, p_partkey

U6

select n_name, sum(l_extendedprice * (1 – l_discount)) as revenue
select n_name, SUM(s_acctbal)
from supplier, partsupp, nation, customer, orders, lineitem, region
where ps_suppkey = s_suppkey and s_nationkey = n_nationkey and (n_name='ARGENTINA') and (s_acctbal > 2000)
group by n_name