Incorporating Intersection 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

Abhinav Jaiswal

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

June, 2022

Declaration of Originality

I, **Abhinav Jaiswal**, with SR No. **04-04-00-10-42-20-1-18011** hereby declare that the material presented in the thesis titled

Incorporating Intersection 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 **2020-2022**.
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: 30-06-2022 Student Signature

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

Advisor Name: Prof. Jayant R. Haritsa Advisor Signature

DEDICATED TO

My Friends and Family

Acknowledgements

First of all, I would like to express my sincere gratitude to my project advisor, Prof. Jayant R. Haritsa for giving me an opportunity to work on this project. I am thankful to him for his valuable guidance and moral support. His immense knowledge and plentiful experience have encouraged me in all the time of my academic and daily life.

I am very thankful to Anupam Sanghi for providing me the unparalleled guidance. The conversations with him have always brought me more clarity and understanding regarding my project problems. I am also thankful to Kapil Khurana for his mentoring on several occasions.

I also want to thank all my lab mates specially Aman Sachan and Mukul Sharma for their invaluable comments and constructive criticism, which made my thesis stronger and more precise.

Finally, I would also like to thank the Indian Institute of Science and the Department of Computer Science and Automation for providing an excellent study environment. The learning experience has been really wonderful here.

Abstract

Queries in the database application often appear in a hidden form. Further, encryption or obfuscation may have been used to secure the application logic. Hidden Query Extraction (HQE) is a new variant of query reverse-engineering problem where the ground-truth query is additionally available but in a hidden form. This problem has diverse use cases ranging from resurrecting legacy code to query rewriting. To address this problem, a tool named 'UNMASQUE' follows the active-learning extraction algorithms to expose a basal class of hidden warehouse queries.

In the real world, queries containing the set operators are common and they are also present across the various benchmarks. The intersection operator is one of the widely used set operators. Therefore, if the hidden query contains the Intersection operator, the existing HQE tool cannot extract the query correctly. This project will extract the hidden queries containing the Intersection operator under certain assumptions. The extraction process will use some of the existing modules of UNMASQUE as a black box. It will also overcome the challenges faced by 'UNMASQUE' in the case of the Intersection query. Furthermore, potent optimizations have been done to reduce the extraction overheads.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In Database Applications, queries are often present in a hidden form, and the encryption and obfuscation may further secure the application logic. These types of queries are known as hidden queries, and the executable of these queries is denoted by $\mathcal{E}$. *Hidden Query Extraction* (HQE) has recently been introduced to extract these types of queries. It is a novel variation of the *Query Reverse Engineering* (QRE) problem where the ground-truth query is also available in a hidden form that is not easily accessible. The generic problem tackled in QRE is as follows: Given a database instance $\mathcal{D}_\mathrm{i}$ and a populated result $\mathcal{R}_\mathrm{i}$, identify a candidate SQL query $\mathcal{Q}_\mathrm{c}$ such that $\mathcal{Q}_\mathrm{c}(\mathcal{D}_\mathrm{i}) = \mathcal{R}_\mathrm{i}$ [4].

HQE can be defined as follows: *Given a black-box application $\mathcal{A}$ containing a hidden query $\mathcal{Q}_H$ (in either SQL format or its imperative equivalent), and a Database instance $\mathcal{D}_I$ on which $\mathcal{A}$ produces a populated result $\mathcal{R}_I$, unmask $\mathcal{Q}_H$ to reveal the original query (in SQL format)* [3].

The presence of the hidden ground-truth query has several advantages: (i) The output query is no longer dependent on the initial ($\mathcal{D}_\mathrm{I}$, $\mathcal{R}_\mathrm{I}$) instance. (ii) Since the application executable can be executed repeatedly on different databases, efficient and focused procedures can be designed for precisely identifying $\mathcal{Q}_\mathrm{H}$.

Database applications widely use the queries containing the Intersection operator. Therefore, if the hidden query contains the Intersection operator, the existing HQE tool (i.e., UNMASQUE) cannot extract the query correctly because it will be out of its extractable domain class.

We will extract the hidden Intersection queries containing SPJ *(Select, Project, Join)* clauses under the certain assumptions. In the case of Intersection query, SPJ is widely used and present across the various benchmarks. We have checked the TPC-H [2] and TPC-DS [1] benchmark, in which we found that none of the Intersection queries contains GAOL (*Group by, Aggregate, Order By and Limit*) clauses in the sub-queries. Currently, we are handling SPJ clause and

Figure 1.1: $\mathcal{Q}_{\mathrm{H}} = \mathcal{Q}_{\mathrm{U}} \cap \mathcal{Q}_{\mathrm{L}}$

GAOL has been marked as future work. In the case of a Hidden Intersection query, the query template we will handle for $\mathcal{Q}_{\mathrm{H}}$ is shown in Figure 1.1. The $\mathcal{Q}_{\mathrm{H}}$ template can be extended for multiple Intersection operator in a similar manner.

In Figure 1.1, P_{EX} denotes the projected columns, T_{EX} denotes the set of extracted tables, J_{EX} denotes the equi-join predicates, F_{EX} denotes the Filter predicates and 'X' denotes either 'U' or 'L' where 'U' represents upper sub-query and 'L' represents lower sub-query of $\mathcal{Q}_{\mathrm{H}}$.

Organization

In Chapter 2, we will discuss the background of the existing HQE tool and briefly describe some of the modules we have used in our work with modifications. We will also discuss the challenges faced by the existing tool in the case of the Hidden Intersection query. Chapter 3 will discuss the problem statement and the assumptions to address the problem. Chapter 4 will discuss the overview of our proposed work, and Chapter 5 will discuss the architecture of the extraction process in detail. The experimental results on various aspects of the query extraction process are discussed in Chapter 6. Finally, we have discussed the conclusions and future work in Chapter 7.

Chapter 2

UNMASQUE

UNMASQUE is a platform-agnostic HQE tool that uses a variety of methods to reveal the hidden query Q_H via active-learning i.e., by examining the outputs of application executions on specially built Database instances. It uses a sophisticated combination of *Database Mutation* and *Database Generation* strategies to extract the hidden queries.

Currently, UNMASQUE can extract a substantial class of SPJGAOL *(Select, Project, Join, Group By, Aggregate, Order By and Limit)* under certain assumptions [4]. The UNMASQUE pipeline is shown in Figure 2.1 in which *Database Mutation* strategies are used to extract the SPJ clauses from the queries, whereas *Database Generation* strategies are required for the subsequent clauses (GAOL)[4]. The Query class that UNMASQUE can extract is defined as *Extractable Query Class (EQC)*. It is defined as (i) All filter predicates are present on the non-key columns and of the form *column op value*. Further, for numeric columns, $op \in \{=, <, >, \leq, \geq, between\}$, whereas for textual columns, $op \in \{=, like\}$; (ii) Join graph is a sub-graph of schema graph comprised of all valid PK-FK and FK-FK edges.

Currently, UNMASQUE cannot extract the hidden query containing the Intersection operator. Therefore, if the hidden query Q_H is of the form shown in Figure 1.1, in that case, UNMASQUE will either extract the wrong query or throw an error.

2.1 UNMASQUE modules

Now, we will discuss some of the UNMASQUE modules that we have used in our work with modifications.

2.1.1 From Clause Extractor

The following procedure is applied in order to identify whether a table T is present in Q_H or not: Pick a table T and temporarily rename it to *temp*. Next, run the executable $\mathcal{E}$ on this

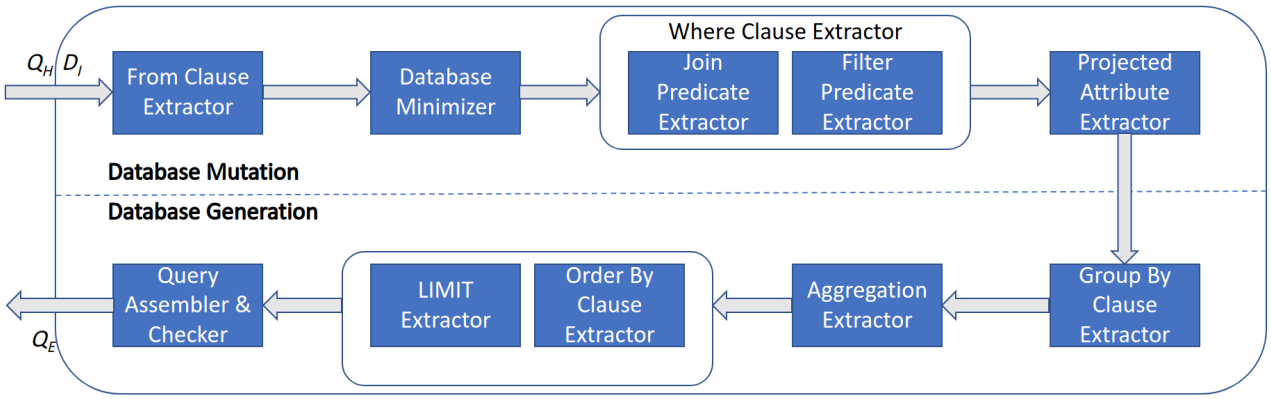


Figure 2.1: UNMASQUE Pipeline

mutated schema; if the database engine immediately throws an error, then the table T is part of the hidden query; else, the execution will terminate after a short timeout period. At last, the renamed table *temp* reverted to its original name. Perform this operation iteratively for all the tables present in database instance $\mathcal{D}_{\mathrm{I}}$.

2.1.2 Database Minimizer

The database size for the enterprise database applications can be very large. Consequently, running the $\mathcal{Q}_{\mathrm{H}}$ multiple times on a large database during the extraction process may take an excessive amount of time. As a result, the UNMASQUE pipeline relies heavily on the *Database Minimizer* module due to the performance point of view. So, *Database Minimizer* is an important module used in the UNMASQUE pipeline. It addresses the row-minimality problem, which is defined as: *Given a Database instance $\mathcal{D}_I$ and an executable $\mathcal{E}$ producing a populated result $\mathcal{R}_I$ on $\mathcal{D}_I$, derive a reduced Database instance $\mathcal{D}_{min}$ from $\mathcal{D}_I$ such that removing any row from any table present in $\mathcal{Q}_H$ results in empty or null output* [4].

To produce $\mathcal{D}_{\mathrm{min}}$ from $\mathcal{D}_{\mathrm{I}}$ the following elementary procedures are applied by the *Database Minimizer*: Pick a table T from the set of extracted tables that contains more than one row, and divide it roughly into two halves. Run $\mathcal{E}$ on the first half, and if the output result is populated, retain the first half. Otherwise, retain only the second half.

In the case of queries belonging to the *EQC* class, there always exists a $\mathcal{D}_{\mathrm{min}}$ denoted by $\mathcal{D}^1$, where each table in T_{E} contains only a single row.

Case	$\mid R_1 \mid = \phi$	$\mid R_2 \mid = \phi$	Predicate Type	Action Required
1	No	No	$i_{min} \leq A \leq i_{max}$	No Predicate
2	Yes	No	$l \leq A \leq i_{max}$	Find l
3	No	Yes	$i_{min} \leq A \leq r$	Find r
4	Yes	Yes	$l \leq A \leq r$	Find l and r

Figure 2.2: Filter Predicate Cases

2.1.3 Filter Predicates Extractor

We iteratively check each set of (non-key) columns present in T_{E}, for its presence in a filter predicate (note that as per EQC class each such attribute can appear in at most one filter predicate), Similarly for Intersection query each attribute can appear in at most one filter predicate of each sub-query. For ease of presentation, we will explain this module in context of integer columns.

Numeric Predicates: Let [$\mathrm{i_{min}}$, $\mathrm{i_{max}}$] be the value range of column A's integer domain, and assume a range predicate $\mathrm{l} \leq \mathrm{A} \leq \mathrm{r}$, where l and r need to be identified. Note that all the comparison operators ($=, <, >, \leq, \geq$, between) can be represented in this generic format – for eg., $\mathrm{A} < 25$ can be written as $\mathrm{i_{min}} \leq \mathrm{A} \leq 24$. To check for presence of a filter predicate on column A, we first create a ${D_{\mathrm{min}}}^{\mathrm{mut}}$ instance by replacing the value of A in one row with $\mathrm{i_{min}}$ in D_{min} , then run $\mathcal{E}$ and get the result – call it $\mathrm{R_1}$. We get another result – call it $\mathrm{R_2}$ – by applying the same process with $\mathrm{i_{max}}$. Now, the existence of a filter predicate is determined based on one of the four disjoint cases shown in Figure 2.2. If the match is with Case 2 (resp. 3), we use a binary-search-based approach over ($\mathrm{i_{min}}$, a] (resp. [a, $\mathrm{i_{max}}$)), to identify the specific value of l (resp. r), where a is the value of column A that is present in D_{min} . After this search completes, the associated predicate is added to the filter predicate. Finally, Case 4 is a combination of Cases 2 and 3, and can therefore be handled in a similar manner. Since the value of only one column (say t.A) is changed at a time, it ensures that any change in the result is solely due to the change in t.A. This enumerative method ensures that we correctly identify filter predicates of the type column op value with op $\in (=, <, >, \leq, \geq$, between) for each numeric database column [3].

2.1.4 Join Predicate Extractor

To extract the key-based equi-join predicates in the $\mathcal{Q}_{\mathrm{H}}$, it begins with SG, the schema graph of the original database composed of all semantically valid key connecting edges. They generate an (undirected) induced sub-graph from SG, where vertices represent the key columns in t_{E}, and edges represent potential join connections between them. The sub-graph is then transitively

closed into a set of cliques using the transitive property of inner equi-joins. Finally, by keeping one of the fundamental n-length cycles (n = number of nodes in the clique), each clique is turned into a cycle graph, hereafter referred to as a cycle. The candidate join-graph is the collection of cycles. They pick a cycle CYC from the candidate join-graph and remove a pair of edges (e_1, e_2) which results in partitioning the cycle into two new connected components. Then, the new components are converted back into smaller cycles (CYC_1 and CYC_2) by reintroducing the relevant missing edge. Next, it will negate all the column values in $\mathcal{D}_{\text{min}}$ corresponding to the vertices present in CYC_1. After that, it will run $\mathcal{E}$ on this mutated database and check for the result; if the result is empty, it concludes that at least one of the edges, either e_1 or e_2 is present in the join-graph. So, both e_1 and e_2 are returned to the parent cycle CYC; otherwise, CYC_1 and CYC_2 are included as new candidates in the candidate join-graph.

2.1.5 Projected Attribute Extractor

The projected columns in the *Select* clause can appear in various forms – native database columns, renamed columns, aggregated columns, and computed columns. In order to extract these columns, this module treats each result column as an (unknown) constrained scalar function of one or more database columns. It identifies the scalar function, assuming linear dependence on the column variables. Let O denote the visible output column, and A, B be the (unknown) database columns that may affect O. Under the assumption of linearity, the function connecting A and B to O can be expressed with the following equation structure:

$$aA + bB + cAB + d = O$$

where a, b, c and d are constant coefficients. With this framework, the extraction process proceeds with identifying the dependency list, which establishes the identities of A and B, followed by the function identification, which computes the values of a, b, c and d.

2.2 Challenges

Now, we will discuss the challenges faced by some of the UNMASQUE modules in the case of hidden Intersection queries.

(i) *Table mapping problem*: The *From Clause Extractor* is only able to extract the set of tables present in the $\mathcal{Q}_{\text{H}}$, but will not be able to map the extracted tables to their respective sub-query in $\mathcal{Q}_{\text{H}}$.

(ii) *Incorrect $\mathcal{D}_{min}$:* When the *Database Minimizer* tries to minimize the given Database instance $\mathcal{D}_{\text{I}}$, it will produce the $\mathcal{D}_{\text{min}}$ on which $\mathcal{E}$ will produce empty result, and the UNMASQUE pipeline will not proceed further. This problem may occur in the case of Intersection query in

which an attribute having mutually disjoint filter range predicate may present across all the sub-queries. Therefore, we need to find another way to compute $\mathcal{D}_{\min}$ such that it results in the populated output.

(iii) *Additional Filter predicates:* When the *Filter Predicates Extractor* tries to extract the predicates present in the $\mathcal{Q}_{\mathrm{H}}$, it will extract the additional predicates consisting of the '=' operator on the attributes that are part of the *Select* clause of the hidden query. When the extractor mutates the attributes' value present in the *Select* clause, the output result of the $\mathcal{E}$ will be empty due to its corresponding projected attribute in another sub-query of $\mathcal{Q}_{\mathrm{H}}$.

(iv) *Incorrect Filter predicates*: If the $\mathcal{Q}_{\mathrm{U}}$ and $\mathcal{Q}_{\mathrm{L}}$ contains the overlapping filter predicate on the same attribute, then filter extractor may extract the subsumed predicate on that attribute. For eg. *c_acctbal* < 4000 in $\mathcal{Q}_{\mathrm{U}}$ and *c_acctbal* between 1000 and 5000 in $\mathcal{Q}_{\mathrm{L}}$, then the extracted predicate will be *c_acctbal* between 1000 and 4000. The extracted predicate is incorrect because there is no lower bound on attribute *c_acctbal* in $\mathcal{Q}_{\mathrm{U}}$ but the extracted predicate is putting a lower bound of 1000.

(v) *Mapping of Filter predicates:* The *Filter Predicates Extractor* is only able to extract the set of predicates present in the $\mathcal{Q}_{\mathrm{H}}$, but will not be able to map the extracted predicates to their respective sub-query in $\mathcal{Q}_{\mathrm{H}}$.

Chapter 3

Problem Framework

This chapter will discuss the problem statement and the underlying assumptions of our proposed solution.

Problem Statement:

Given a hidden query $\mathcal{Q}_H$ containing the Intersection operator, and a database instance $\mathcal{D}_I$ on which $\mathcal{Q}_H$ produces populated result $\mathcal{R}_I$, unmask $\mathcal{Q}_H$ to reveal the original query.

Assumptions

In this work, we will extract the hidden Intersection queries containing SPJ (*Select, Project, Join*) clauses under the following assumptions:

1. Attributes having the *Filter* predicates should not be the part of the *Select* clause.
2. There must be an attribute present in all the sub-queries having mutually disjoint filter range predicate.
3. All the tables present in the sub-queries must be joined via equi-join.
4. Each key column can appear in at most one equi-join predicate p. However, that p can present across multiple sub-queries. We can see these types join predicates on the databases of type star schema.

Table 3.1: Notations

Symbol	**Meaning**	**Symbol**	**Meaning**
$\mathcal{E}$	Application Executable	$\mathcal{D}_{\min}$	Minimized database
$\mathcal{Q}_H$	Hidden Query	$\mathcal{D}_I$	Database instance
$\mathcal{Q}_E$	Extracted Query	$\mathcal{D}_I^{mut}$	Mutated database instance
$\mathcal{Q}_U$	Upper sub-query	$\mathcal{D}_{min}^{mut}$	Mutated $\mathcal{D}_{\min}$
$\mathcal{Q}_L$	Lower sub-query	v	Attribute's domain value
$\mathcal{Q}_{EU}$	Extracted Upper sub-query	SG	Schema Graph of $\mathcal{D}_{\mathrm{I}}$
$\mathcal{Q}_{EL}$	Extracted Lower sub-query	EQC	Extractable Query Class
$\mathcal{D}^t$	Database with at most t rows in tables of $\mathcal{Q}_{\mathrm{E}}$	C	Connected components list

Chapter 4

Solution Overview

This chapter will give an overview of our extraction approach on the hidden Intersection query $\mathcal{Q}_H$ and the database instance $\mathcal{D}_I$, shown in Figure 4.1 and Figure 4.2, respectively. The Intersection extraction pipeline is shown in Figure 4.3.

```
select c_mktsegment as segment
from customer, nation
where c_acctbal <3000 and c_nationkey = n_nationkey
and n_name= 'BRAZIL'
intersect
select c_mktsegment
from customer, nation, orders
where c_acctbal between 4000 and 5000 and c_nationkey = n_nationkey
and c_custkey = o_custkey and o_orderdate < '1995-03-15'
```

Figure 4.1: Hidden Query $\mathcal{Q}_H$

Customer

c_custkey	c_nationkey	c_acctbal	c_mktsegment
1	2	2500	BUILDING
2	1	1000	FURNITURE
3	2	4500	BUILDING
4	3	2458	MACHINERY

Nation

n_nationkey	n_name
1	FRANCE
2	BRAZIL
3	CANADA

Orders

o_orderkey	o_custkey	o_orderdate
8	2	1996-03-02
9	1	1995-04-07
10	3	1994-02-10

ε

Output

c_mktsegment
BUILDING

Figure 4.2: Database Instance $\mathcal{D}_I$

Firstly, we will input the given $\mathcal{Q}_H$ and $\mathcal{D}_I$ to the existing *From Clause Extractor* module of UNMASQUE. This module will extract the set of tables T_E present in $\mathcal{Q}_H$. Furthermore, the extracted set extracted tables T_E and $\mathcal{D}_I$ will be given as input to our *Extended DB Minimizer* module.

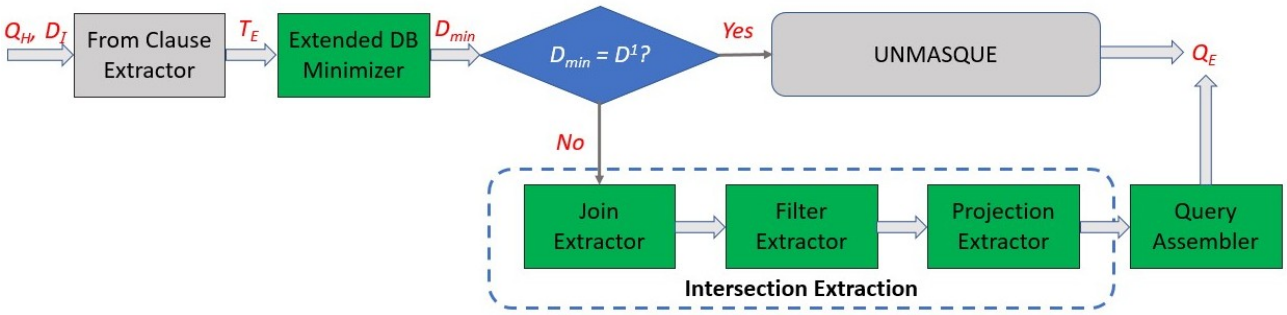


Figure 4.3: Intersection Pipeline Architecture

This module follows an algorithm that performs binary halving on each table in T_E. After each halving, it checks the output result of by running the executable $\mathcal{E}$. If it gets the populated result on any of the halves, it retains that half; otherwise, it further partitions both the halves into two more halves, and checks for the combination that produces the populated result. The output produced by this module on $\mathcal{D}_I$ is shown in Figure 4.4.

Customer

c_custkey	c_nationkey	c_acctbal	c_mktsegment
1	2	2500	BUILDING
3	2	4500	BUILDING

Nation

n_nationkey	n_name
2	BRAZIL

Orders

o_orderkey	o_custkey	o_orderdate
10	3	1994-02-10

$\mathcal{E}$

Output

c_mktsegment
BUILDING

Figure 4.4: $\mathcal{D}_{\min}$ on $\mathcal{D}_I$

After getting the $\mathcal{D}_{\min}$, we will check the cardinality of each table T present in the $\mathcal{D}_{\min}$. If any of the tables contain two rows (*because of our assumption* 2), the pipeline proceeds with our *Join Extractor* module; otherwise, by assuming that the $\mathcal{Q}_H$ belongs to the *EQC*, it will call the UNMASQUE pipeline.

The *Join Extractor* module takes $\mathcal{D}_{\min}$ and the schema graph of the database as input. The schema graph contains the vertices and edges where each vertex represents a key column, and an edge represents the linkage between a pair of key columns. This module will use the *Join Predicate Extractor* module of UNMASQUE, which will extract all the equi-join predicates present in the $\mathcal{Q}_H$.

Nation

n_nationkey	n_name
2	BRAZIL

Customer

c_custkey	c_nationkey	c_acctbal	c_mktsegment
1	2	2500	BUILDING
3	2	4500	BUILDING

Orders

o_orderkey	o_custkey	o_orderdate
10	3	1994-02-10

Figure 4.5: Mapping of rows in equi-join predicates

The extracted join predicates will not contain any information about the mapping to their respective sub-queries in the $\mathcal{Q}_{\mathrm{H}}$. Therefore, the *Join Extractor* module will extract this information in the following manner: For each pair of attributes present in the extracted equi-join predicates, it mutates the attribute's value on a tuple by tuple basis, and finds the mapping of tuples between the pair of attributes, as shown in Figure 4.5.

Now, we will replicate the tuple of the tables consisting of only one row in the $\mathcal{D}_{\mathrm{min}}$, and getting joined with all the tuples of the tables having two rows (same pair of attributes) as shown in Figure 4.6.

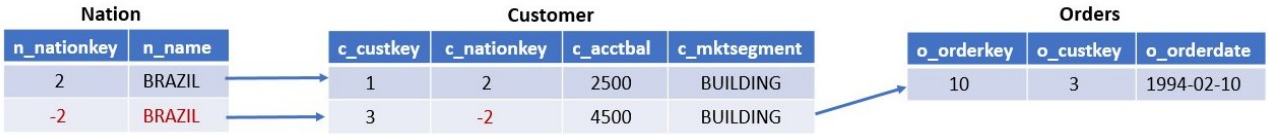
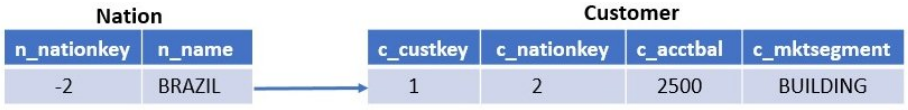


Figure 4.6: Connected Components

In this Figure, the *Nation* table consists of only one row in the $\mathcal{D}_{\mathrm{min}}$ and getting joined with all the rows of the *Customer* table on the same pair of attributes (i.e., *n_nationkey*, *c_nationkey*). So, we will replicate the tuple in the *Nation* table and change its value to v on the *n_nationkey* attribute. Furthermore, we will also change the value of any one mapped rows in the Customer table on attribute *c_nationkey* by the same value v (i.e., *attibute's domain value*).

Replication of tuples will form the two connected components graph from a single component, where each vertex denotes a tuple, and an edge denotes the connection between a pair of tuples. Each of the connected components will work as $\mathcal{D}^1$ for a sub-query in the $\mathcal{Q}_{\mathrm{H}}$, and will help in mapping all the extracted tables and predicates to their respective sub-queries.

After getting the connected components, we will use the existing *Filter Predicates Extractor* module to extract all the filter predicates present in the $\mathcal{D}_{\mathrm{min}}$, and map these extracted predicates to their respective sub-query using the component graph.



Projection: c_mktsegment as segment, **From:** nation, customer **Filter:** c_nationkey = n_nationkey, c_acctbal <= 2999, n_name = 'BRAZIL'

Figure 4.7: Components corresponding to $\mathcal{Q}_{EU}$

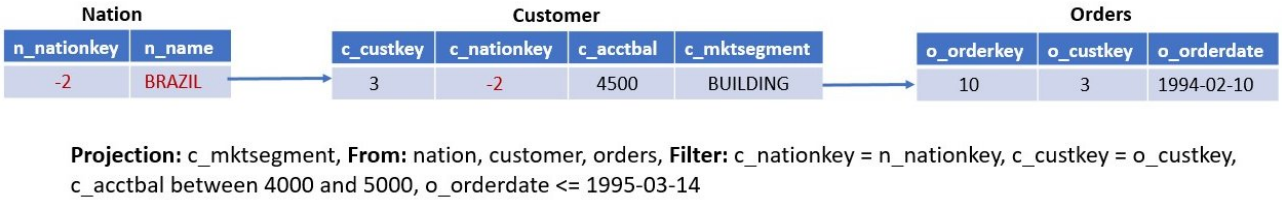


Figure 4.8: Components corresponding to $\mathcal{Q}_{EL}$

In the case of the Intersection query, each of the columns in the *Select* clause of $\mathcal{Q}_{\mathrm{H}}$ will be present in the extracted filter predicate containing the '=' operator.

It is due to the mapping to their corresponding column in the *Select* clause of other sub-queries. The *Projected Columns* modules will mutate each such predicates' column and check its effect on the output. If it still gets the populated result, it will add those predicates' columns to the *Select* clause of their respective component. The extracted predicates and the projected columns is shown in Figure 4.7 and Figure 4.8. At last, the *Query Assembler* module will assemble all the components and extracts the hidden query $\mathcal{Q}_{\mathrm{H}}$.

Chapter 5

Unmasking Intersection

This chapter will discuss in detail about the modules that we have used in the *Intersection Pipeline Architecture* in order to extract the hidden Intersection query.

5.1 Extended DB Minimizer

$\mathcal{D}_{\text{min}}$ for the Intersection query

In the case of a hidden Intersection query, the maximum cardinality of a table present in $\mathcal{D}_{\text{min}}$ depends on the number of Intersection operators present in the $\mathcal{Q}_{\text{H}}$. In this section, we will attempt to substantiate our assertion.

Lemma 1: *Existence of $\mathcal{D}^{k+1}$ for the queries containing k Intersection operators.*
Proof. We will be using the fact that the query belonging to EQC must have a single-row database $\mathcal{D}^1$ such that it produces a populated result (proved in [3]). As in the case of queries containing intersection operators, each of the sub-queries will lie in the domain of EQC. So, if there are k Intersection operators, there will be $k+1$ sub-queries. Based on $\mathcal{D}^1$ for EQC, the $\mathcal{D}_{\text{min}}$ should be $\mathcal{D}^{k+1}$, but the same tuple can satisfy more than one sub-query. As per our assumption, there should be at least one column with a mutually disjoint range predicate in each of the sub-queries. Therefore, that column should have $k+1$ different values in the $\mathcal{D}_{\text{min}}$, leading to a table of size $k+1$. Hence, the $\mathcal{D}_{\text{min}}$ will be $\mathcal{D}^{k+1}$.

This module is an extended version of the *Database Minimizer* which can produce $\mathcal{D}_{\text{min}}$ even if the $\mathcal{Q}_{\text{H}}$ contains the Intersection operator. It takes a set of extracted tables T_{E} and the database instance $\mathcal{D}_{\text{I}}$ as input. The problem with the earlier minimizer was that if it does not get the populated result on the first half of the table, it implicitly assumes that the result will be in the second half, which was true for the queries belonging to the EQC class. In the case

of the Intersection query, the minimum tuples required from a table T to produce a populated result can lie across both the table halves. So explicitly, we need to check for the second half's result if the table's first half fails to produce the populated result.

Algorithm 1: *Extended DB Minimizer*

```
Data: T_E, Q_E and D_I
Result: D_min
foreach table T in T_E do
    while |T| > 1 do
        Divide T into two halves T_U and T_L
        T ← T_U
        if E(D_I^mut) ≠ φ then
            Drop T_L
        else
            T ← T_L
            if E(D_I^mut) ≠ φ then
                Drop T_U
            else
                Divide T_U into two halves T_U1 and T_U2
                Divide T_L into two halves T_L1 and T_L2
                T ← T_Ui T_Lj ;                          // i, j ∈ {1, 2}
                if ∃ (i, j) s.t. E(D_I^mut) ≠ φ then
                    Drop T_U and T_L
                    if |T| = 2 then
                        break
                else
                    Partition Table
                end
            end
        end
        if |T| = 1 then
            break
        end
    end
end
```

The *Extended DB Minimizer* works in the following manner to produce the $\mathcal{D}_{\text{min}}$: It picks a table T from T_{E}, and divides it into roughly two halves (*i.e.,* $\{T_U,\ T_L\}$). Next, it runs the $\mathcal{E}$ on the first half, and if the output result is populated, it retains the first half; otherwise, it runs $\mathcal{E}$ on the second half. If the output result on the second half is populated, it retains the second half; otherwise, performs binary halving on both the halves of table such that, it divides them into further two halves (*i.e.,* $\{T_{U1},\ T_{U2}\}$ *for* T_U *and* $\{T_{L1},\ T_{L2}\}$ *for* T_L). Then,

checks the populated result on all the possible four combinations (*i.e.,* $\{T_{U1}\ T_{L1}\}$, $\{T_{U1}\ T_{L2}\}$, $\{T_{U2}\ T_{L1}\}$ *and* $\{T_{U2}\ T_{L2}\}$) as T. If the $\mathcal{Q}_{\mathrm{H}}$ contains only one Intersection operator, in that case, the maximum cardinality of a table in $\mathcal{D}_{\min}$ will be 2. Therefore, each of these tuples is guaranteed to lie among one of the possible four combinations. The minimizer algorithm will pick a combination that produces the populated result and reduce the table T to that selected combination.

Algorithm 2: *Partition Table*

```
Data: T
if |T| ≤ 2(k + 1) then
    foreach row r in T do
        Delete r from T
        if E(D_I^mut) = φ then
            Restore r into T
        end
    end
    break
else
    Divide T into 2(k + 1) parts
    foreach part p in T do
        Delete p from T
        if E(D_I^mut) = φ then
            Restore p into T
        end
    end
end
```

The above approach will work for the $\mathcal{Q}_{\mathrm{H}}$ containing at most one Intersection operator, but if the $\mathcal{Q}_{\mathrm{H}}$ contains more than one Intersection operator, then all the possible combinations may fail to produce the populated result. In that case, the minimizer algorithm will call the *Partition Table* algorithm that will reduce the size of the table T to half in the worst case. This algorithm assumes that there will be at most k Intersection operator in the $\mathcal{Q}_{\mathrm{H}}$. Based on this assumption, the algorithm divides the table T into $2(k+1)$ parts and iteratively checks the requirement of each part in order to get the populated output. If the output result is still populated while discarding any part, discard it; otherwise, it retains that part.

Time Complexity

Let $T_{\mathrm{i}}, T_{\mathrm{i+1}}, ...T_{\mathrm{m}}$ be the set of tables present in T_{EU} and $T_{\mathrm{j}}, T_{\mathrm{j+1}}, ...T_{\mathrm{n}}$ in T_{EL}. τ_{m} denotes $|T_{\mathrm{i}}| * |T_{\mathrm{i+1}}| * .... * |T_{\mathrm{m}}|$ (i.e., the product of table sizes) and τ_{n} denotes $|T_{\mathrm{j}}| * |T_{\mathrm{j+1}}| * .... * |T_{\mathrm{n}}|$

where $|T|$ is the cardinality of table T.

We have used τ (i.e., $\tau_{\mathrm{m}} + \tau_{\mathrm{n}}$) to denote the time taken by the application to operate on the original database instance $\mathcal{D}_{\mathrm{I}}$ on which it produces the result $\mathcal{R}_{\mathrm{I}}$. Consider the case when the hidden query $\mathcal{Q}_{\mathrm{H}}$ contains at most one Intersection operator. In the worst case of the algorithm, the table T will be present in both $\mathcal{Q}_{\mathrm{U}}$ and $\mathcal{Q}_{\mathrm{L}}$ (i.e., T_{EU} and T_{EL} are the same), and the algorithm needs to execute the $\mathcal{E}$ at most six times in order to reduce the size of the table T to its half (i.e., $(\tau_{\mathrm{m}}/2 + \tau_{\mathrm{n}}/2)$. Therefore, the time complexity of the algorithm in the worst case will be 6*($\tau/2 + \tau/4$ +...+1) i.e., O(τ).

Consider the case when the $\mathcal{Q}_{\mathrm{H}}$ contains more than one Intersection operator, in this case the minimizer need to execute the executable $\mathcal{E}$ for at most $2k$ times in a single iteration, which guarantees to reduce the size of table T to its half in the worst case. Hence, the time complexity of the *Partition Table* algorithm in the worst case will be $2k$*($\tau + \tau/2 + \tau/4$ +....1) i.e., O($k\tau$).

Therefore, the overall time complexity of the *Extended DB Minimizer* algorithm will be O($k\tau$).

5.2 Join Extractor

This module will extract all the equi-join predicates present in the $\mathcal{Q}_{\mathrm{H}}$, and map these extracted predicates to their respective sub-queries. It starts by taking the schema graph SG as input, which would be an induced subgraph on the original schema graph of the database. In the SG, vertices are the key columns in the T_{E}, whereas each edge denotes the join linkage between a pair of key attributes. This algorithm will use the existing *Join Predicate Extractor* module of the UNMASQUE to extract the set of equi-join predicates present in the $\mathcal{Q}_{\mathrm{H}}$ [3]. However, it will not extract any information about mapping these predicates to their respective sub-queries. In the case of an Intersection query, the same equi-join predicate may present across the different sub-queries. Therefore, to overcome this problem, we will find the mapping information of each tuples, present across the tables whose attributes are the part of equi-join predicate. We will get this information by mutating the attributes of equi-join predicates on a tuple basis.

For each pair of attributes (*i.e.,* (a_x, a_y)) present in the extracted join predicates, we will check the cardinality of their respective table (*i.e.,* (t_x, t_y))and based on that we will mutate the values. These are the possible combinations of tuples present across (t_x, t_y):

Case 1: *t_x and t_y has only one row*

In this case, both the rows (*i.e.,* (r_x^i, r_y^j)) are getting joined with each other. This case is automatically handled by the *Join Predicate Extractor* module.

Algorithm 3: *Join Extractor*

Data: $\mathcal{D}_{\text{min}}$ and SG
$JG_E \leftarrow$ Join Predicate Extractor C $\leftarrow \phi$ count $\leftarrow 0$
foreach *(*a_x*,* a_y*) in* JG_E **do**
  if $|t_x| > |t_y|$ *;* `// similary for` $|t_x| < |t_y|$
  then
    foreach *row* r_x^i *in* t_x *;* `//` $i \in$ `(0, 1)`
    do
      Mutate r_x^i on a_x
      if $\mathcal{E}(\mathcal{D}_{min}^{mut}) = \phi$ **then**
        C $\leftarrow$ C $\cup$ (r_x^i, r_y^j) ; `// j = 0`
        count $\leftarrow$ count + 1
      end
      restore t_x
    end
    if *count = 2* **then**
      Replicate r_y^j in t_y as r_y^{j+1}
      For any value of $i \in (0, 1)$
      Mutate r_y^{j+1} on a_y and r_x^i on a_x by v
      Replace (r_x^i, r_y^j) in C by (r_x^i, r_y^{j+1})
    end
  else if $|t_x| = |t_y|$ *and* $|t_x| = 2$ **then**
    foreach *row* r_x^i *in* t_x **do**
      foreach *row* r_y^j *in* t_y **do**
        if $a_x = a_y$ *in* (r_x^i, r_y^j) **then**
          Mutate a_x and a_y to a value v
          if $\mathcal{E}(\mathcal{D}_{min}^{mut}) \neq \phi$ **then**
            C $\leftarrow$ C $\cup$ (r_x^i, r_y^j)
          end
          Restore t_x and t_y
        end
      end
    end
  end
end
foreach *(*a_x*,* a_y*) in* JG_E **do**
  if $|t_x| = |t_y|$ *and* $|t_x| = 1$ **then**
    C $\leftarrow$ C $\cup$ (r_x^i, r_y^j)
  end
end

Case 2: *Both t_x and t_y have two rows*
In this case, we compare the pair of attributes present in the equi-join predicates on a tuple basis. If the values are the same, we mutate their values simultaneously to a new value v and run the $\mathcal{E}$; if the output result is populated, it means they are getting joined with each other. We will restore the values of the tuples.
Case 3: *t_x has two rows while t_y has only one row (vice-versa)*
We will mutate the join attribute of the table that has two rows. In this case, t_x has two rows, so that we will mutate its attribute a_x on a tuple basis. For each tuple r_x^i, we will mutate a_x and check the output result of $\mathcal{E}$; if it is empty, it means r_x^i is getting joined by r_y^j. We will restore the tuple value.

On attribute (a_x, a_y), if both the rows in t_x are getting joined by the row in table t_y, then we will replicate the tuple present in t_y. After replication, we will change the attribute a_y with a value v on the replicated tuple and will also change a_x for any one tuple in the t_x by the same value. It results in the formation of two connected components from the one component where each component works as $\mathcal{D}^1$ for the sub-queries present in the $\mathcal{Q}_{\mathrm{H}}$. With the help of connected components, we will map all the extracted join predicates to their respective sub-queries.

Time Complexity

Let E be the set of edges in the schema graph SG and C^{key} denotes the set of key columns in T_{E}. The time complexity of the *Join Predicate Extractor* module to extract the equi-join predicate will be $\mathrm{O}(E * |C^{key}|^2)$ discussed in [3]. Consider the number of distinct equi-join predicates present in the hidden Intersection query as n. For each extracted equi-join predicate, this algorithm mutates the key-column and executes the $\mathcal{E}$ on mutated $\mathcal{D}_{\min}$ which will take constant time. So, the overall time complexity of this algorithm will be $\mathrm{O}(n + E * |C^{key}|^2)$.

5.3 Filter Extractor

This module will use the existing *Filter Predicate Extractor* module of UNMASQUE [3]. It will take the extracted list of connected component as input where each component represents in which vertices represents the tuples and edge represents the connection between the tuples. So, we will extract predicates on each component and map them accordingly to their respective sub-queries.

5.4 Projection Extractor

This module will extract all the attributes in the *Select* clause of sub-queries present in $\mathcal{Q}_{\mathrm{H}}$. It takes a refined filter list RFL as input consists of all the Filter predicates having '=' operator.

It also requires the information about the maximum cardinality of the tables present in $\mathcal{D}_{\text{min}}$, to find the number of *Select* clause present in $\mathcal{Q}_{\text{H}}$. We have shown the *Projection Extractor* algorithm for *one* Intersection operator i.e., the maximum cardinality of the tables will be *two*.

Algorithm 4: Projection Extractor

```
Data: RFL, C
Result: y = x^n
PL ← φ
output ← E(D_min)
foreach elt in output do
    L ← φ
    foreach p in RFL do
        if p.val = elt then
            L ← L ∪ p
        end
    end
    foreach p1, p2 in L do
        Choose a different value for p1, p2's attribute
        if E(D_min^mut) ≠ φ and v ∈ output then
            PL ← PL ∪ (p1.att, p2.att)
            break
        end
        Restore p1, p2's attribute
    end
end
```

To identify all the attributes in the *Select* clause of $\mathcal{Q}_{\text{H}}$, the following elementary procedure is applied: For each attributes in the populated result of $\mathcal{D}_{\text{min}}$, it extracts all the predicates from RFL whose value is same as attribute's value and stores them in an empty list L. It picks two predicates at a time from list L, and change the values of both predicate's attribute to a value v in their respective component. Next, it runs the executable $\mathcal{E}$ and checks the output result; if it is populated and v is present in the output, then it adds the selected predicates' attribute to the projection list of their respective component; otherwise, choose another pair and repeat the process. At last, it will return the Projection List PL containing all the attributes present in the *Select* clause and we can map these predicates using the connected components.

Time Complexity

Let us assume that the number of attributes present in the output of the hidden query is c, and the number of predicates present in the RFL is n. So for each output column, we need to once iterate the RFL. Therefore, it will be of $O(cn)$.

In the worst case, the size of list L will be $O(n)$, when all the predicates have the same value. So we need to perform $\binom{N}{2}$ operations to find the correct pairs of predicates present in the *Select* clause of Q_{H}, and we will do this operation for each of the columns present in the output. So, it will be $O(cn^2)$. Therefore, the overall time complexity of the *Projection Extractor* algorithm will be $O(cn^2)$.

Chapter 6

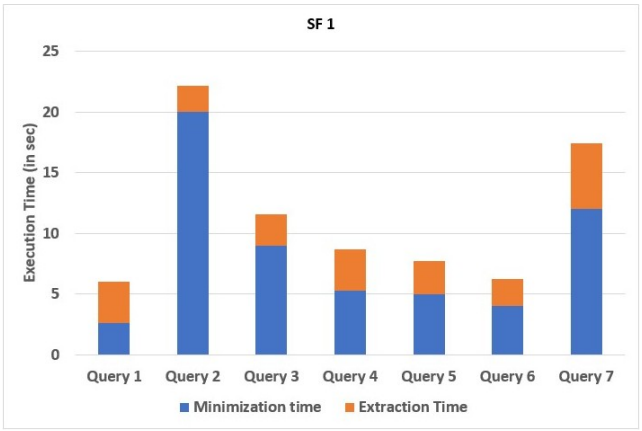
Experiments

The proposed intersection extractor is implemented in Python 3.6 and integrated with the existing UNMASQUE codebase. We have broadened the extractable domain of Hidden Query Extraction. Our experiments are carried out on a vanilla PostgreSQL 11 database platform (Intel Xeon 2.3 GHz CPU, 32GB RAM, 3TB Disk, Ubuntu Linux) with default primary-key indices. We have reported the extraction overhead to unmask the hidden queries containing the intersection operator. To conduct a better evaluation, we need complexity in queries that TPC-H and TPC-DS benchmarks provide. These complex queries are derived from the TPC-H and TPC-DS benchmarks so that all of our assumptions hold.

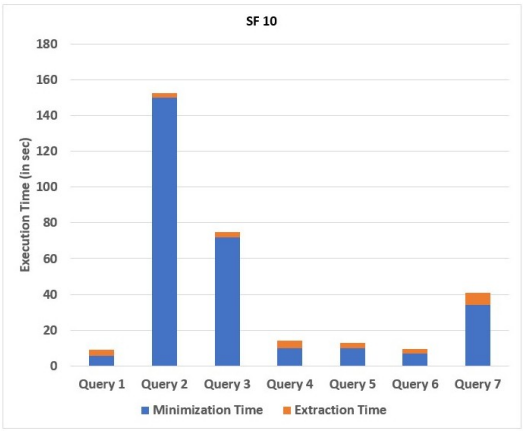
All these derived benchmark queries are listed in the appendix. The total end-to-end time taken to extract each of the seven queries on a 1 GB and 10 GB initial instance (with a populated result) is shown in Figure 6.1a and 6.1b. The first three queries are derived from the TPC-H benchmark, and the other four are derived from the TPC-DS benchmark. In addition, the breakup of the Intersection extractor module and the Minimization time is also shown in the figure.

We have done the manual verification of all the output extracted queries. The extraction times are practical for offline analysis environments. When we drilled down into the performance profile, intersection extraction time was independent of the initial database $\mathcal{D}_{\mathrm{I}}$ size. This module operates with a miniscule database, so the extraction overhead is less. By the definition of $\mathcal{D}_{\mathrm{min}}$, the Intersection extractor module time for both 1GB and 10GB database instances will similar. Our extraction process is database scale independent.

We are internally sampling the database tables before running Extended DB Minimizer. Sampling is a non-deterministic process where sometimes few tables are left unsampled. If a



(a) Extraction Time Comparison on 1GB



(b) Extraction Time Comparison on 10GB

Figure 6.1: Comparison of Minimization Time vs Extraction Time

table left unsampled, then the minimizer goes for the full copy of the table in order to minimze the database. Sampling is the reason for the variable nature of Minimization time. As a case in point, Query 7 on a 1GB database instance has the second largest Minimization time, whereas Query 3 on a 10GB database instance has the second largest Minimization time.

Chapter 7

Conclusion and Future Work

In this work, we have extracted the hidden query $\mathcal{Q}_{\mathrm{H}}$ containing the Intersection operator, which will extend the scope of the existing HQE tool (i.e., UNMASQUE). Earlier, the UNMASQUE could only handle the *Union* among the set operators. We have also extended the working of *Database Minimizer* such that it will produce $\mathcal{D}^k$ instead of $\mathcal{D}^1$ in the case of the Intersection query. The extraction process of the Intersection query is independent of the database sizes as the detection of the Intersection query is confirmed only after the *Extended DB Minimizer* module.

We have extracted the Intersection query only in the case of the SPJ clause as they are widely used, and in the future, we will try to extend its scope for the GAOL clause. We will also try to relax our assumption for the extraction of the Intersection query.

Bibliography

[1] *TPC-DS*. www.tpc.org/tpcds/, . 1

[2] *TPC-H*. www.tpc.org/tpch/, . 1

[3] K. Khurana and J. Haritsa. Opaque query extraction. technical report. https://dsl.cds.iisc.ac.in/publications/report/TR/TR-2021-02.pdf, 2021. Indian Institute of Science. 1, 5, 14, 17, 19

[4] 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, 3, 4

Appendix

Hidden Query $\mathcal{Q}_\mathbf{H}$

Query 1

select c_mktsegment as segment
from customer,nation
where c_acctbal < 3000 and c_nationkey = n_nationkey and n_name = 'BRAZIL'
intersect
select c_mktsegment
from customer,nation,orders
where c_acctbal between 1000 and 5000 and c_nationkey=n_nationkey and c_custkey = o_custkey and n_name = 'ARGENTINA';

Query 2

select o_orderstatus, o_totalprice
from customer,orders
where c_custkey = o_custkey and o_orderdate < date '1995-03-10'
intersect
select o_orderstatus, o_totalprice
from lineitem, orders
where o_orderkey = l_orderkey and o_orderdate > date '1995-03-10' and l_shipmode = 'AIR';

Query 3

select p_container,p_retailprice,ps_availqty
from part,supplier,partsupp
where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_brand='Brand45'
intersect
select p_container,p_retailprice,ps_availqty
from part,supplier,partsupp
where p_partkey = ps_partkey and s_suppkey=ps_suppkey and p_brand='Brand15' and p_size

> 10;

Query 4

select cs_quantity,cs_wholesale_cost
from catalog_sales,customer_demographics
where cs_bill_cdemo_sk = cd_demo_sk and cs_sales_price <50 and cd_education_status = 'College'
intersect
select cs_quantity,cs_wholesale_cost
from catalog_sales,customer
where cs_bill_customer_sk = c_customer_sk and cs_sales_price between 70 and 150;

Query 5

select c_first_name,c_last_name
from customer, customer_address
where c_current_addr_sk = ca_address_sk and c_birth_year < 1950 and c_birth_country = 'ICELAND'
intersect
select c_first_name,c_last_name
from customer, customer_demographics
where c_current_cdemo_sk = cd_demo_sk and c_birth_year between 1956 and 1996 and cd_education_status = 'College';

Query 6

select c_first_name, ca_country
from customer_address,customer,date_dim
where ca_address_sk = c_current_addr_sk and d_date_sk = c_first_sales_date_sk and c_birth_country = 'AUSTRALIA'
intersect
select c_last_name,ca_country
from customer_address, customer,date_dim
where ca_address_sk = c_current_addr_sk and d_date_sk = c_first_sales_date_sk and c_birth_country = 'HUNGARY';

Query 7

select cs_quantity, cs_wholesale_cost, d_day_name, c_first_name
from catalog_sales, customer, date_dim
where cs_bill_customer_sk = c_customer_sk and d_date_sk = c_first_sales_date_sk and cs_sales_price < 50

intersect
select cs_quantity, cs_wholesale_cost, d_day_name, c_first_name
from catalog_sales, customer, date_dim
where cs_bill_customer_sk = c_customer_sk and d_date_sk = c_first_sales_date_sk and cs_sales_price between 70 and 150 and d_year between 1998 AND 1998 + 2;

Extracted Query $\mathcal{Q}_\mathbf{E}$

Query 1
Select c_mktsegment as segment
From customer, nation
Where n_nationkey = c_nationkey and c_acctbal $\leq$ 2999.0 and n_name = 'BRAZIL'
Intersect
Select c_mktsegment
From customer, nation, orders
Where n_nationkey = c_nationkey and c_custkey = o_custkey and c_acctbal between '1000.0' and '5000.0' and n_name = 'ARGENTINA';
Query 2
Select o_orderstatus, o_totalprice
From customer, orders
Where c_custkey = o_custkey and o_orderdate $\leq$ '1995-03-09'
Intersect
Select o_orderstatus, o_totalprice
From lineitem, orders
Where o_orderkey = l_orderkey and l_shipmode = 'AIR' and o_orderdate $\geq$ '1995-03-11' ;
Query 3
Select p_retailprice, ps_availqty, p_container
From part, partsupp, supplier
Where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_brand = 'Brand15' and p_size $\geq$ 11
Intersect
Select p_retailprice, ps_availqty, p_container
From part, partsupp, supplier
Where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_brand = 'Brand45';
Query 4
Select cs_wholesale_cost, cs_quantity

From catalog_sales, customer_demographics
Where cd_demo_sk = cs_bill_cdemo_sk and cs_sales_price ≤ 49.0 and cd_education_status = 'College'
Intersect
Select cs_wholesale_cost, cs_quantity
From catalog_sales, customer
Where c_customer_sk = cs_bill_customer_sk and cs_sales_price between '70.0' and '150.0';

Query 5

Select c_last_name, c_first_name
From customer, customer_address
Where c_current_addr_sk = ca_address_sk and c_birth_year ≤ 1949 and c_birth_country = 'ICELAND'
Intersect
Select c_last_name, c_first_name
From customer, customer_demographics
Where c_current_cdemo_sk = cd_demo_sk and c_birth_year between '1956' and '1996' and cd_education_sta
= 'College';

Query 6

Select ca_country, c_last_name
From customer, customer_address, date_dim
Where c_current_addr_sk = ca_address_sk and c_first_sales_date_sk = d_date_sk and c_birth_country = 'HUNGARY'
Intersect
Select ca_country, c_first_name
From customer, customer_address, date_dim
Where c_current_addr_sk = ca_address_sk and c_first_sales_date_sk = d_date_sk and c_birth_country = 'AUSTRALIA';

Query 7

select cs_quantity,cs_wholesale_cost,d_day_name, c_first_name
from catalog_sales, customer, date_dim
where cs_bill_customer_sk = c_customer_sk and d_date_sk = c_first_sales_date_sk and cs_sales_price ≤ 49.0
Intersect
select cs_quantity, cs_wholesale_cost, d_day_name, c_first_name
from catalog_sales, customer, date_dim

where cs_bill_customer_sk = c_customer_sk and d_date_sk = c_first_sales_date_sk and d_year between ‘1998’ and ‘2000’ and cs_sales_price between ‘70.0’ and ‘150.0’;