Extracting Hidden Algebraic Predicates

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

BY

Aman Sachan

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

June, 2022

Declaration of Originality

I, **Aman Sachan**, with SR No. **04-04-00-10-42-20-1-18094** hereby declare that the material presented in the thesis titled

Extracting Hidden Algebraic Predicates

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2020-22**.
With my signature, I certify that:

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

Date: June 2022

Student Signature

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

Advisor Name: Prof. Jayant R. Haritsa

Advisor Signature

DEDICATED TO

My Family and Friends

for their love and support

Acknowledgements

I am deeply grateful to Prof. Jayant R. Haritsa for his unmatched guidance, enthusiasm and supervision. He was always been a source of inspiration for me. I have been extremely lucky to work with him.

I am thankful to Anupam Sanghi for his assistance and guidance. It had been a great experience to work with him. My sincere thanks goes to my fellow lab mates and seniors especially Kapil Khurana, Abhinav Jaiswal, Mukul Sharma and Sumang Garg for all the help and suggestions. Also I thank my friends who made my stay at IISc pleasant, and for all the fun we had together.

Finally, I am indebted with gratitude to my parents and brother for their love and inspiration that no amount of thanks can suffice. This project would not have been possible without their constant support and motivation.

Abstract

Queries in database applications can be hidden due to encryption or dense imperative code, making the query challenging to reveal. The Hidden Query Extraction (HQE) problem was first defined in [4], and to address this problem, they have created a tool called UNMASQUE (**U**nified **N**on-invasive **MA**chine for **S**ql **QU**ery **E**xtraction). The diverse use-cases for this problem range from resurrecting legacy code to query rewriting. UNMASQUE non-invasively extracts the hidden SQL queries in database systems using an active-learning approach. It's a lightweight procedure that is application and platform independent.

At this time, UNMASQUE cannot extract various SQL constructs like Algebraic Predicates. Algebraic Predicates are the predicates of type $\langle column_1\ operator\ column_2 \rangle$ where $operator \in \{=, <, \leq, >, \geq\}$, and $column_1$ and $column_2$ can be of the same table (i.e., *intra-table predicates*), as well as of different tables (i.e., *inter-table predicates* that consist of *Equi-Joins, Non-Equi Joins*). In this work, we have expanded the extractable domain of UNMASQUE by successfully extracting the queries containing Algebraic Predicates.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The new query reverse-engineering problem of unmasking hidden SQL queries termed HQE (Hidden Query Extraction) was recently introduced in [4]. A ground-truth query is provided here but in a hidden form that is hard to access. For example, the original query may be explicitly hidden in a black-box application executable. Moreover, encryption or obfuscation may have been incorporated to further protect the application logic. An alternative scenario is that the application is visible but effectively opaque because it is comprised of hard-to-comprehend SQL (such as those arising from machine-generated object relational mappings), or poorly documented imperative code that is not easily decipherable. Such "hidden executable" situations could also arise in the context of legacy code, where the source has been lost or misplaced over time, or when third-party proprietary tools are part of the workflow, or if the software has been inherited from external developers. More formally, the HQE problem is: *Given a black-box application* $\mathcal{A}$ *containing a hidden query* Q_H *(in either SQL format or its imperative equivalent), and a database instance* D_I *on which* $\mathcal{A}$ *produces a populated result* R_I*, unmask* Q_H *to reveal the original query (in SQL format). That is, we intend to find the precise* Q_H *such that* $\forall i$, $Q_H(D_i) = R_i$.

UNMASQUE (**U**nified **N**on-invasive **MA**chine for **S**ql **QU**ery **E**xtraction) is a platform-independent hidden query extractor used to address HQE Problem. UNMASQUE operates in a sequential pipeline manner shown in Figure 1.1. It employs a judicious combination of *database mutation* and *synthetic database generation* to extract a basal set of queries containing the essential SPJGAOL (**S**elect, **P**roject, **J**oin, **G**roup By, **A**ggregate, **O**rder By, **L**imit) clauses. More specifically, single-block equi-join conjunctive queries expressible in the form:

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

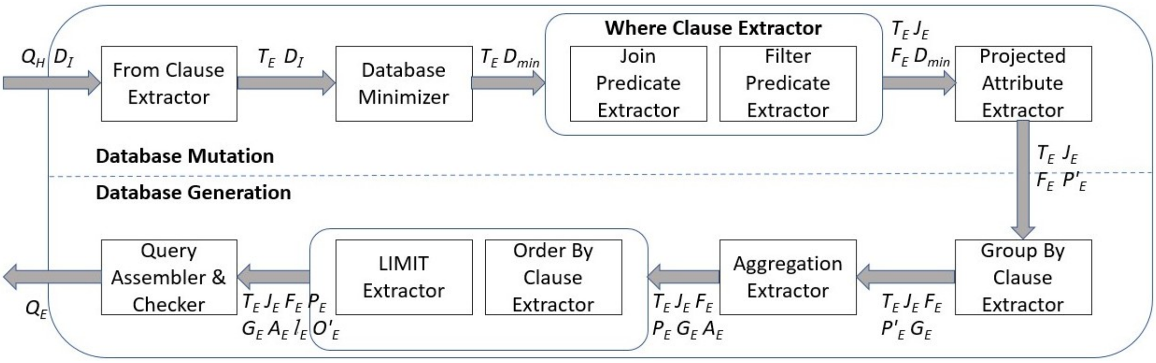


Figure 1.1: UNMASQUE Architecture

1.1 Motivation

Algebraic Predicates are of the form $\langle column_1\ operator\ column_2\rangle$ where $operator \in \{=, <, \leq, >, \geq\}$, and $column_1$ and $column_2$ can be of the same table (i.e., *intra-table predicates*), as well as of different tables (i.e., *inter-table predicates* that consist of *Equi-Joins, Non-Equi Joins*). Algebraic Predicates are commonly and widely used. Also, it is evident from the various enterprise-class benchmarks like TPC-H [3] and TPC-DS [2]. In the TPC-H benchmark, 20 out of 22 queries have Algebraic Predicates, whereas around 15% of them have intra-table predicates. One of them is shown in Figure 1.2:

```
Select l_shipmode, count(*) as count
From Orders, Lineitem
Where o_orderkey = l_orderkey
    and l_shipdate ≤ l_commitdate
    and l_commitdate ≤ l_receiptdate
    and l_receiptdate between '1994-01-01' and '1995-01-01'
    and l_extendedprice ≤ o_totalprice
    and l_extendedprice ≤ 70000
    and o_totalprice ≥ 60000
Group By l_shipmode
Order By l_shipmode
```

Figure 1.2: Exemplar Query

1.2 Technical Challenges

UNMASQUE treats a column as having an upper bound (UB) and a lower bound (LB), and both of them must be concrete. In other words, each column is treated as having an Arithmetic Predicates, i.e., *column op value* where $op \in \{=, <, \leq, >, \geq, between\}$ for numeric columns and

$op \in \{=, like\}$ for textual columns. As the exemplar query contains algebraic predicates, it will produce a wrong result or fail in between due to the generation of incorrect filter predicates. In contrast to this, two major problems need to be handled:

- *Variable Bounds:* In the case of algebraic predicates, as the value of the columns varies, the bounds also vary, i.e., for different database instances, we will be getting different bounds. For example, $Column_1 < Column_2$ exists, where the value for $Column_1$ and $Column_2$ turns out to be $Value_1$ and $Value_2$, respectively. As $Column_1 < Column_2$, $Value_1$ can't be greater or equal to $Value_2$. But when the $Value_2$ changes, the range of $Value_1$ also changes, i.e., the bounds of $Column_1$ change with the change in the value of $Column_2$.

- *Multiplicity of Bounds:* Also, more than one predicate in the hidden query Q_H per column leads to multiple lower and upper bounds, where the bounds are variable. For example, $Column_1 < Column_2$ and $Column_1 < Column_3$ exist, where the value for $Column_1$, $Column_2$ and $Column_3$ turns out to be $Value_1$, $Value_2$ and $Value_3$, respectively. As $Column_1 < Column_2$ and $Column_1 < Column_3$, $Value_1$ can't be greater or equal to $Value_2$ and $Value_3$, i.e., $Value_1$ can't be greater or equal to whichever is minimum among $Value_2$ and $Value_3$. So, here $Value_2$ and $Value_3$ will act as two different bounds on $Column_1$.

1.3 Contribution

This work successfully extracted the queries containing the *Algebraic Predicates* that covers a variety of constructs like *intra-table predicates* and *inter-table predicates* which further consists of *Equi-Joins*, and *Non-Equi Joins* between any pair of columns. The key design principles that help attain the desired objective are by considering the column's value, manipulating their values, and observing any changes in the bound of other columns – these principles are discussed in detail in Chapter 4 and 5. We have evaluated the implemented module's behavior on a suite of complex decision-support queries. The performance results of these experiments, conducted on a vanilla PostgreSQL [1] platform, indicate that module precisely identifies the Algebraic Predicates in our workloads in a timely manner. Also, there will be negligible overhead in the case of queries without algebraic predicates and obtain better results in the case of equi-join extraction between key columns.

Organization The remainder of this report is organized as follows: In Chapter 2, we have described the background of UNMASQUE, which is required to explain further work. The problem framework is discussed in Chapter 3. Further, the key design principles of our work are highlighted in Chapter 4, and then described in detail in Chapter 5. The experimental framework and performance results are reported in Chapter 6. Finally, our conclusions and future research avenues are summarized in Chapter 7.

Chapter 2

Prerequisites

In this chapter, we will only be discussing the modules required to explain the further work, i.e., Database Minimizer and Where Clause Extractor (consists of Equi-Join and Filter Predicates Extractor) of the UNMASQUE.

2.1 Database Mimimizer

D_I is likely to be huge for enterprise database applications, and therefore repeatedly executing $\mathcal{E}$ on this extensive database during the extraction process may take an impractically long time. To tackle this issue, they minimize the database such that each table in T_E contains only a single row. To identify a D^1, they use an iterative-reduction process, i.e., pick a table t from T_E containing more than one row and divide it roughly into two halves. Run $\mathcal{E}$ on the first half, and if the result is populated, retain only this first half. Otherwise, retain only the second half, and eventually, all the tables in T_E have been reduced to a single row by this process.

2.2 Where Clause Extractor

It consists of two sub-modules: (i) Join Predicate Extractor and (ii) Filter Predicate Extractor. Join Predicate Extractor finds all the inner equi-joins between key columns, and Filter Predicate Extractor finds the Arithmetic Predicates on non-key columns. It is explained in more detail in the further sections.

2.2.1 Join Predicate Extractor

To extract the key-based equi-join predicates, they start with SG, the original schema graph of the database comprised of all semantically valid key-connecting edges. From SG, they create an (undirected) induced subgraph whose vertices are the key columns in T_E, and edges are the potential join linkages between these columns. Then, using the transitive property of inner equi-joins, this subgraph is converted through transitive closure into a collection of cliques. Finally, each clique is converted to a cycle graph, hereafter referred to as a cycle, by retaining one of the elementary n-length cycles (n = number of nodes in the clique). Note that in this case, even the trivial elementary graph with $n = 2$ (a pair of nodes and an edge between them) is also considered to be a cycle. The complete collection of cycles is referred to as the candidate join-graph. After that, they will remove a pair of edges to make it disconnected and run the executable. If it produces a populated result, they will discard the join condition corresponding to the removed edge, and two new cycles will be created (one from each of the disconnected components); otherwise, at least one edge will be a viable candidate for equi-join. They negate the values corresponding to those columns and verify them. Repeat the process till no more edges can be removed.

2.2.2 Filter Predicate Extractor

They will assume that all non-key columns in C_A are potential candidates for the filter predicates in Q_H. In here, we have described the process for integer columns but can extend the same logic for other data types. Let $[i_{min}, i_{max}]$ be the value range of column A's integer domain, and assume a range predicate $l \leq A \leq r$, where l and r need to be identified. Note that all the comparison operators $(=, <, >, \leq, \geq, between)$ can be represented in this generic format – for example, $A < 25$ can be written as $i_{min} \leq A \leq 24$. To check for a filter predicate on column A, they first create a D^1_{mut} instance by replacing the value of A with i_{min} in D^1, then run $\mathcal{E}$ and get the result – call it $R_{i_{min}}$. They get another result – call it $R_{i_{max}}$ – by applying the same process with i_{max}. Now, the existence of a filter predicate is determined based on one of the four disjoint cases shown in Table 2.1.

Table 2.1: Filter Predicate Cases

Case	$\lvert \boldsymbol{R_{i_{min}}} \rvert = \phi$	$\lvert \boldsymbol{R_{i_{max}}} \rvert = \phi$	**Predicate Type**	**Action Required**
1	False	False	$i_{min} \leq A \leq i_{max}$	No Predicate
2	True	False	$l \leq A \leq i_{max}$	Find l
3	False	True	$i_{min} \leq A \leq r$	Find r
4	True	True	$l \leq A \leq r$	Find l and r

If the match is with Case 2 (resp. 3), they use a binary-search-based approach over $(i_{min}, a]$ (resp. $[a, i_{max})$) to identify the specific value of l (resp. r), where a is the value of column A that is present in D^1. Finally, Case 4 is a combination of Cases 2 and 3 and can be handled similarly. They apply the above procedure for each of the non-key columns in C_A. 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 arithmetic predicates are correctly identified for each non-key numeric database column.

Chapter 3

Problem Framework

In this chapter, we summarize the basic problem statement, and the underlying assumptions of our solution.

Statement: The hidden query Q_H contains Algebraic Predicates, unmask Q_H to reveal the original query such that $\forall i$, $Q_H(D_i) = R_i$.

Assumptions: We can handle a substantial class of queries termed as Extractable Query Class (EQC$^+$). These are the following assumptions for EQC$^+$:

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

In addition to this, we are retaining the assumptions from [4] required for other modules.

Table 3.1: Notations

Symbol	**Meaning**
$\mathcal{E}$	Application Executable
Q_H	Hidden Query
Q_E	Extracted Query
T_E	Set of tables in Q_E
C_A	Set of columns in T_E
C_H	Set of columns that are part of Join and Filter Predicates in Q_H
D_I	Initial Database Instance
D^1	Database with one row in T_E
D_{mut}	Mutated database
J_E	Set of Join predicates
F_E	Set of Filter predicates

Chapter 4

Solution Overview

We will discuss the extraction process of hidden queries containing Algebraic Predicates, i.e., the predicates of type *column op column* where $op \in \{=, <, \leq, >, \geq\}$ for numeric columns and op is $=$ for textual columns. At the same time, the Arithmetic Predicates (i.e., *column op value* where $op \in \{=, <, \leq, >, \geq, between\}$ for numeric columns and $op \in \{=, like\}$ for textual columns) can also be a part of the hidden query. Furthermore, we will be discussing for $\{=, \leq, \geq\}$ and can extend the same logic for $\{<, >\}$. The architecture is shown in Figure 4.1.

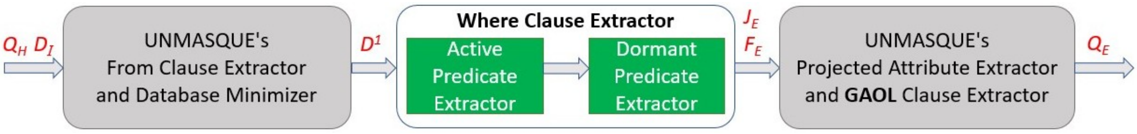
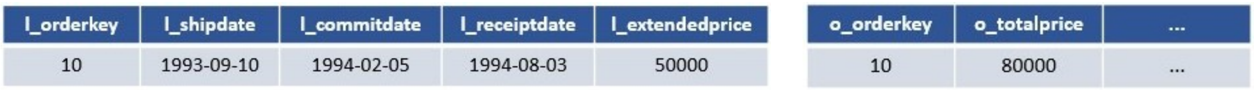


Figure 4.1: Updated UNMASQUE Architecture

Let's consider the tweaked TPC-H query mentioned in Chapter 1. Here we will discuss the 'Where Clause' extraction, and the rest of the clauses will be handled by the original UNMASQUE discussed in [4] and [5]. Till *Where Clause Extraction*: *From Clause Extractor* will give all the tables present in the hidden query, i.e., *Orders* and *Lineitem*, and the *Database Minimizer* will reduce D_I to D^1 (shown in Figure 4.2).



l_orderkey	l_shipdate	l_commitdate	l_receiptdate	l_extendedprice
10	1993-09-10	1994-02-05	1994-08-03	50000

(a) Lineitem table

o_orderkey	o_totalprice	...
10	80000	...

(b) Orders Table

Figure 4.2: Reduced Database, D^1

Initially, we will assume that all columns (C_A) are potential candidates for the filter predicates in Q_H (including the key-columns) and make a graph G in which each vertex represents a column from C_A. The graph G for the running example is shown in Figure 4.3.

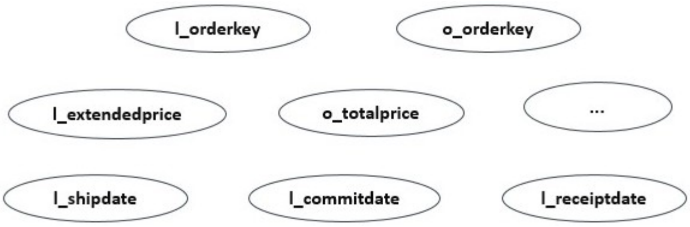


Figure 4.3: Graph, G

To find whether the column is a part of any filter predicates, we will use the UNMASQUE's *Filter Predicate Extractor*. It will provide concrete bounds for all such columns that are part of the *Where Clause* in Q_H, but the bounds will depend on D^1. We will distinguish the predicates present in Q_H in the following types: (i) If there is a predicate $col \leq col'$ with col has an upper bound (i.e., $col \leq val$) and col' has a lower bound (i.e., $col' \geq val'$) such that the value of col in D^1 is less than val' and the value of col' in D^1 is greater than val. Hence, the predicate $col \leq col'$ will show no effect on the result and bounds while doing a single mutation, and such predicates are termed *Dormant Predicates.* (ii) All the remaining ones will be termed *Active Predicates*. One point to note here is that predicate classification is based on D^1. For different D^1, some of the active predicates may become dormant, and some of the dormant predicates may become active. Nevertheless, the definition for classification will remain intact in terms of D^1.

In the running example, $l\_extendedprice \leq o\_totalprice$ is a *Dormant Predicates*, whereas all other comes under *Active Predicates.* Firstly, we will do the screening (i.e., removing all the columns that don't have any filter predicates) by picking a column and try to find the UB and LB using the similar approach as discussed in Chapter 2.2.2. If for both i_{min} and i_{max} it produces a populated result signifying that there is no presence of predicate on that column and will remove the corresponding vertex from the graph. In our case, the bounds came out to be *"o_orderkey = 10 and l_orderkey = 10 and l_shipdate ≤ 1994-02-05 and l_commitdate between 1993-09-10 and 1994-08-03 and l_receiptdate between 1994-02-05 and 1995-01-01 and l_extendedprice ≤ 70000 and o_totalprice ≥ 60000".* Now, we will find all the *Active Predicates* and then jump to *Dormant Predicates.* To find *Active Predicates*, we will find all such columns whose value in D^1 is same as the bound i.e., draw a dashed edge from A to B if A's upper bound is same as B's value and draw a dashed edge from B to A if A's lower bound is same as B's value. The graph after screening and updation is shown in Figure 4.4.

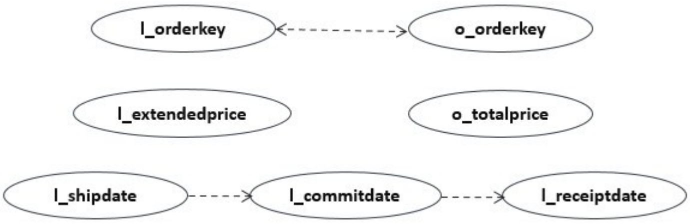


Figure 4.4: G after screening

There can be two types of edges possible: (i) *double arrow-headed* and (ii) *single arrow-headed*. Double arrow-headed edge represents a possible candidate for equality algebraic predicate, and to identify such predicates, we will change both column values in D^1 and run the executable. The equality holds between those columns if mutated D^1, i.e., D^1_{mut} produces a populated result. If yes, merge both nodes; otherwise, remove the dashed edges. In our case, it turns out to be $l\_orderkey = o\_orderkey$. Hence, we will merge the nodes for *l_orderkey* and *o_orderkey*. For single arrow-headed, we will validate them by manipulating the column's value and suppose the predicate (i.e., the cause of the edge) also reflects the changes. In that case, we will make that edge solid, signifying this relation or algebraic predicate is present in the hidden query Q_H.

When we make an edge solid, we recursively find the bounds for the new column and assign them the extreme value. After that, come back again to the original column and then find the next lower and upper bound and repeat the process. In our case, when we find the next bound for *l_receiptdate*, we will recursively assign the columns as i_{min}. Assigning *l_shipdate* with i_{min} will produce a populated result. Same for *l_commitdate*, but when we assign *l_receiptdate*, it will generate an empty result means there are some other bounds also present on *l_receiptdate*. We will conduct a binary search over (i_{min}, *1994-08-03*] to find the predicate $l\_receiptdate \geq$ *1994-01-01*. Now, the updated graph will look as shown in Figure 4.5.

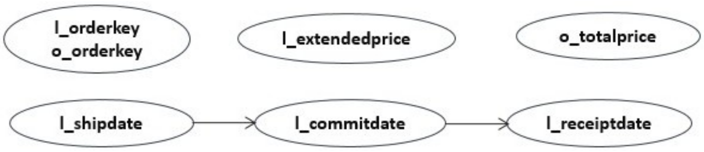


Figure 4.5: G with *Active Predicates*

To find all Dormant Predicates, we will pick a vertex from a connected component in G and assign all its predecessor a minimum value in the topological order. At the same time, all other columns are maintained at the maximum possible value. While choosing the component containing *o_totalprice*, and assigning *o_totalprice* as *60000* leads to an empty result because *l_extendedprice*'s value will be *70000*.

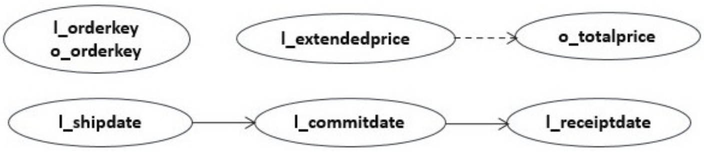


Figure 4.6: Possible *Dormant Predicate*

During the evaluation of *o_totalprice*'s lower bound, it turns out to be same as *l_extendedprice*'s value. Hence, we will add a dashed edge between *o_totalprice* and *l_extendedprice* and validate it by manipulating the *l_extendedprice*'s value. The final graph for the hidden query Q_H is shown in Figure 4.7.

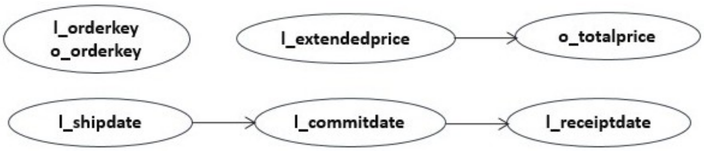


Figure 4.7: Final Graph, G

After the *Where Clause* extraction, the *Projected Attribute Extractor* and *Aggregation Extractor* will identify the *Select Clause*. The *Group By* and *Order By* clauses will be determined by the *Group By Clause Extractor* and *Order By Clause Extractor*, respectively. Now, we will discuss the extraction process in more detail in further chapters.

Chapter 5

Where Clause Extractor

Before jumping to the extraction of the predicates, there are some important points to remember: (i) Concrete bounds on a column refer to the arithmetic predicates present in the hidden query. Also, there can be at most one UB and one LB. If more than one concrete UB exists, then the minimum among them will be the concrete UB, and the rest will be redundant. Similarly, we can say for concrete lower bound. (ii) Suppose some predicate is of the form *Column = Value*, and that column is also used for some other algebraic predicate. In that case, we can replace the column with a value (i.e., RHS of the predicate *Column = Value*) without changing the semantic meaning. We are going to treat such algebraic predicates as arithmetic predicates. As we can recall from Chapter 4, there are two types of predicates. This chapter will discuss the procedure used to identify various predicates.

Lemma 1: *For the* EQC^{+}*, there always exists a* D^1*.*
Proof. Existence of D^1 was proved in [5], and the same can be used for EQC^{+}.

5.1 Active Predicate Extractor

To find the bounds for each column, we will follow the process described in Chapter 2.2.2. This algorithm is similar to the Filter Predicate Extractor of UNMASQUE [4]. The key difference is that they use it on the non-key columns, whereas we have extended it to key and non-key columns. Also, they are using it to identify the concrete filter predicates while we are using it to determine the bounds on the column. We will remove all such vertices for which no bounds are defined, and the remaining vertices correspond to a set of columns C_E.

Time Complexity: Let r denote the range of the column. We require two table updates and two calls to the executable to determine one of the four cases in Figure 2.1, an $O(1)$ operation. If the column has a constraint, we require $\log r$ table updates and corresponding executable calls. Thus, the total time complexity of computing the bounds for a column is $O(\log r)$.

Lemma 2: *For a query in EQC$^+$, C_E will always be same as C_H.*

Proof. To prove this, we first need to prove: (i) $c \in C_H$ iff $\forall t_1, t_2$ such that $R_1 \wedge R_2 = false$, where t_1 and t_2 are the single row database, R_i is $\mathcal{E}(t_i) \neq \phi$, $t_1.c = i_{min}$, and $t_2.c = i_{max}$; (ii) $c \in C_E$ iff $\forall t_1, t_2$ such that $R_1 \wedge R_2 = false$, where t_1 and t_2 are the single row database, R_i is $\mathcal{E}(t_i) \neq \phi$, $t_1.c = i_{min}$, and $t_2.c = i_{max}$.
For (i), let's assume $c \notin C_H$, but $R_1 \wedge R_2 = false$. As $c \notin C_H$, c is unconstrained and can accept all values in the domain (i.e., $i_{min} \leq c \leq i_{max}$). So, at least one t in the domain will have the c's value as i_{min} (and i_{max}), which generates a populated result and vice-versa.
For (ii), let's assume $c \notin C_E$, but $R_1 \wedge R_2 = false$ means D^1 is producing a populated result, but while checking filter predicates for c, both $|R_{imin}|$ and $|R_{imax}|$ turns out to be ϕ. Both are true, so $\exists t_1, t_2$ for which $R_1 \wedge R_2 = true$ and vice-versa. From this, we can conclude for EQC$^+$ that $C_E = C_H$.

Now, we will identify all active predicates present in Q_H based on D^1. More precisely, active predicates are: *Given D^1, the predicate $C \leq X$ is active if X is concrete or X is variable s.t. (i) C's concrete UB doesn't exist, or (ii) X's concrete LB doesn't exist, or (iii) Both C's concrete UB (i.e., $C \leq v_c$) and X's concrete LB (i.e., $X \geq v_x$) exist, then either $v_c > D^1.X$ or $v_x < D^1.C$ satisfy and the same goes for the predicate $C \geq X$ in reverse order. Also, the predicates of type $C = X$, where X can be column or value comes under the class of active predicates.*

Lemma 3: *If a column is a part of some algebraic predicate of the form* ***column op column****, where* ***op*** *in $\{=, <, \leq, >, \geq\}$ then the bounds will be dependent on D^1.*

Proof. Let's consider an algebraic predicate $\langle c_1 \ op \ c_2 \rangle$, where $op \in \{=, \leq, \geq\}$. To prove the lemma, we will cover all the possible cases: (i) For $c_1 = c_2$, both columns should have the same value in D^1 (say val), and while finding the bounds on c_1 (and c_2), it will arise the Case 4 (shown in Table 2.1). We have to find both l and r in this case, and due to $c_1 = c_2$, it will give both l and r as val; (ii) For $c_1 \leq c_2$, while identifying the bounds for c_1 (and c_2), it will lead to Case 3 (and Case 2). It will produce $c_1 \leq val_2$ (and $c_2 \geq val_1$), where val_2 (and val_1) is the value of c_2 (and c_1) in D^1 because c_1's value can not exceed c_2's value. So, if the value fluctuates, the filter predicates will also change accordingly. Similarly, we can show for $c_1 \geq c_2$.

5.1.1 Equality Predicates

Till now, we have computed the bounds for a column, and if the column's lower and upper bound turn out to be same (say *val*), then we will check all such columns whose lower and upper bounds are same and are equal to *val* (i.e., double arrow-headed dashed edge discussed in Chapter 4). We will consider all possible combinations and try to find an equality relation between them, if any exist, by manipulating the value with some other common value and finally merging all the vertices holding an equality relationship. In Algorithm 1, from line 4-11 we have explained the identification of a chain of at most two columns (which means there will be no transitive equality relation between any three columns) and can extend the same for the chain of more than two.

Algorithm 1: Validator

```
   Data: $pred\langle col\ op\ val \rangle$, Set of bounds $\mathcal{B}$
 1 $flag = 0$
 2 foreach $col'$ in $C_E$ do
 3     if $val = D^1.col'$ then
 4         if $op$ is '=' then
 5             Choose a value for both columns
 6             if $Q_H(D^1_{mut} \neq \phi)$ then
 7                 Merge $col$ and $col'$
 8                 Add $\langle col = col' \rangle$ in $F_E$
 9                 $flag = 1$
10             end
11         else
12             Choose a value for $col'$ within $\mathcal{B}$
13             if $\mathcal{B}_{new} \neq \mathcal{B}$ then
14                 Add an edge between $col$ and $col'$
15                 Add $\langle col\ op\ col' \rangle$ in $F_E$
16                 $flag = 1$
17             end
18         end
19     end
20 end
21 if $flag = 0$ then
22     Add $pred$ in $F_E$
23 end
```

Correctness. Due to the presence of $col_1 = col_2 = \cdots$ in the hidden query Q_H, Lemma 2 guarantees that there must be a predicate (i.e., bounds) for col_1, col_2, and $\cdots$. The lower and upper bounds for all columns (i.e., col_1, col_2, and $\cdots$) will turn out to be the same, and the lower bound will be equal to the upper bound signifying the equality predicate (i.e., $column = value$). As all the columns have the same bound, the algorithm will consider all the possible combinations. There will be two possibilities: (i) chosen combination is a proper subset of the columns holding equality relation; (ii) chosen combination is a proper superset of the columns holding equality relation. For (i), while choosing a different value for the columns will produce an empty result due to the left out columns with unmatched values. For (ii), the case will never arise in the first place because we will be checking the combination in increasing order of chain length. So, it already identified the columns having a mutual equality relation.

5.1.2 Inequality Predicates

Now, we will discuss for the active predicates of type *column op X* where $op \in \{<, \leq, >, \geq\}$ and X can be a column or value. We will first validate whether the bound is concrete or variable. The *Validator* (described in Algorithm 1, line 12-17) does it by manipulating the values (within the bounds) one by one of all such columns whose value in D^1 is equal to the bound. If the bound varies, we will make an edge between them; otherwise, we will conclude that the bound is concrete. If the bounds turn out to be variable (due to column *col*), we will iteratively find the new bounds by assigning the col as *min* or *max* depending upon the nature of the bound (if not able to assign, then first find the bounds for *col*). In order to find the bounds, we have assigned the min (and max) for all the columns that form a connected component in graph G, but some of the vertices may remain unexplored. So, we will pick a column and repeat the process.

Correctness. Consider the active predicate of type $C <= X$ (as defined):
(i) X is concrete, and it is the k^{th} UB (i.e., UB$_k$, in terms of D^1): As there can be only one concrete UB, we need to identify all *k-1* variable UB. It is due to the C's transitive relation or direct relation signifying all identified *k-1* columns have a value less than X in D^1. In order to obtain X, we need to assign all *k-1* columns a value greater than X. It is done in the *Do-While* loop of Algorithm 2. First, we have computed the UB (i.e., UB$_1$) for C and validated whether it is concrete or variable. If it is concrete, it's done; otherwise, we will assign the obtained column $\hat{C}$ as maximum and recursively do it for $\hat{C}$ and obtain UB$_2$, and so on. Again come back to C and find the next UB$_i$. If $i < k$, again repeat the process; otherwise, we have reached UB$_k$ and finally obtain X.
(ii) X is variable, and C's concrete UB doesn't exist: By default, $C \leq i_{max}$ exists, but X is a variable UB. It will restrict column C's assignment at some point in time, and from (i), we can

Algorithm 2: Active Predicate Extractor

```
Data: $D^1$
Make a graph $G$, from $C_A$
$F_E = \phi$
foreach col in $C_A$ do
    Find bounds for col
    if $col.UB = i_{max}$ and $col.LB = i_{min}$ then
        Remove col from $G$
    else
        do
            Call Validator
            if variable bounds then
                Update $G$
                Recursively, find the bounds for the connected column
            end
            Find next bound for col
        while bounds doesn't change;
    end
end
```

say it will identify the predicate, $C \leq X$.
(iii) X is variable, and X's concrete LB doesn't exist: We will identify the predicate while processing for X in the reverse direction. By default, $X \geq i_{min}$ exists, but C will be a variable LB. It will restrict column X's assignment at some point, and the same explanation from (i) in the reverse direction will identify the predicate, $C \leq X$.
(iv) X is variable, and both C's concrete UB (i.e., $C \leq v_c$) and X's concrete LB (i.e., $X \geq v_x$) exist, then either $v_c > D^1.X$ or $v_x < D^1.C$ satisfy: For $v_c > D^1.X$ exists, (i) will identify the predicate and for $v_x < D^1.C$, (i) in reverse direction will identify the predicate.
More specifically, all algebraic predicates of type $C_1 \geq C_2$ comes under the type $C \leq X$ where C is C_1 and X is C_2, and the proof for the arithmetic predicate of type $C \geq v$ will follow same as (i).

Time Complexity: The time consumed to create a graph will be $O(|C_A|)$, and after the screening, there will be at most $|C_E|$ nodes present in the graph. If n predicates are present, we require one table update and one call to the executable per predicate, leading to $O(n)$ operation of constant time. In contrast, for every operation, the *Validator* has to check every column in the worst case. Hence, the total time complexity will be $O(n * |C_E|)$.

5.2 Dormant Predicate Extractor

There can be *Dormant Predicates*, as discussed in Chapter 4, and it arises due to the overlapping bounds of two columns present in different component. In contrast, one column decides a variable bound on the other column but has a value greater than the concrete upper bound of the other column and vice-versa. More formally, Given D^1, the predicate $C_1 \leq C_2$ is dormant if both C_1's concrete UB (i.e., $C_1 \leq v_1$) and C_2's concrete LB (i.e., $C_2 \geq v_2$) exist s.t. $v_1 > D^1.C_2$ and $v_2 < D^1.C_1$ satisfy.

To address this problem, we will convert the dormant predicate into active predicate by picking a pair of connected components, assign the maximum possible value for all the columns in one component. Now, we can call Algorithm 2 to find such predicates. In this approach, for n components we have to check for $\binom{n}{2}$ combinations. So, for efficiency, we can assign the maximum for all the columns of the remaining component. It finds the relation between multiple components and reduces the number of iterations to n. Also, in place of calling Algorithm 2, we will assign the minimum possible value to each predecessor of the chosen column in the topological order. *Why should we assign minimum in topological order?* As the graph was built on the less than relationship, we can't assign a column with the minimum possible value without assigning the minimum value to all the columns which follow the less than relationship with that column.

Algorithm 3: Dormant Predicate Extractor

```
Data: Set of bounds $\mathcal{B}$, Graph $G$
$\mathcal{CC} \longleftarrow$ All connected components in $G$
foreach comp in $\mathcal{CC}$ do
    Assign all columns of $\{\mathcal{CC} - comp\}$ as maximum value compliant with $\mathcal{B}$
    Queue $Q \longleftarrow$ topological sort of comp
    foreach col in $Q$ do
        val $\longleftarrow$ Find minimum value for col
        if val not compliant with $\mathcal{B}$ then
            Call Validator
            Update $\mathcal{B}, G, \mathcal{CC}$, and $Q$
        end
    end
    Restore $D^1$
end
```

Lemma 4: *G will be an acyclic graph.*

Proof. Let's assume $\exists F_E$, for which the G is cyclic. It means that the relationship between some of the columns will be of the form $C_i \leq C_j \leq \cdots \leq C_k \leq C_i$, which is semantically equivalent to $C_i = C_j = \cdots = C_k$. We will identify it in Chapter 5.1, and for $C_i, C_j, \cdots, C_k$, there will be a single node in graph G. In other words, a cycle will collapse to a single node.

Correctness. If $v_1 \leq v_2$, then the predicate $C_1 \leq C_2$ is redundant; otherwise, while computing for C_2, we make all the columns as concrete or default upper bound except C_2 and its predecessors. It will violate the condition $V_2 > D^1.C_1$ and makes $C_1 \leq C_2$ an active predicate.

Time Complexity: As there are $|C_E|$ number of nodes, the time required to find all the connected components will be $O(|C_E|)$. In the worst case, there will be $|C_E|$ components present, and for each column we have to update every other column's value. Hence, the time complexity for the algorithm will be $O(|C_E|^2)$.

Chapter 6

Experiments

The new modules were tested against various queries to verify the correctness and see how much overhead was incurred due to the additions. We ran all the experiments on PostgreSQL hosted on an Intel Xeon 3.2 GHz CPU, 32GB RAM, Linux equipped machine. Experiments were performed on slightly tweaked TPC-H benchmark queries such that they lie under EQC^+, and in some cases, Algebraic Predicates were explicitly introduced.

As UNMASQUE extracts the hidden ground truth, it is independent of the original database as long as assumptions are met. As TPC-H provides complexity in queries, it will be a viable option for better evaluation. Additionally, the TPC-H queries test the performance of new modules as part of the UNMASQUE system, rather than just checking the performance of standalone modules. UNMASQUE's original codebase was used as a black-box, other than the change in Where Clause Extractor. The algorithms were implemented in Python 3.6 and have been integrated with the UNMASQUE codebase. We have manually verified all the extracted queries. All the queries used for experiments are listed in the Appendix. The experiments[1] are performed on the TPC-H database of sizes 1, 10, and 100 GB (i.e., SF1, SF10, and SF100).

6.1 Extraction Time wrt DB Size

We have analyzed the extraction time for various queries with different database sizes. Minimization time is directly affected by the database size, so we are considering the post-minimization time. The implemented modules work on D^1 and are independent of the original database instance. So, the extraction time should not depend on the database's size, and the performed experiment (shown in Figure 6.1) also reflects that.

---

[1] Extraction time is considered after the execution of the *Database Minimizer* module.

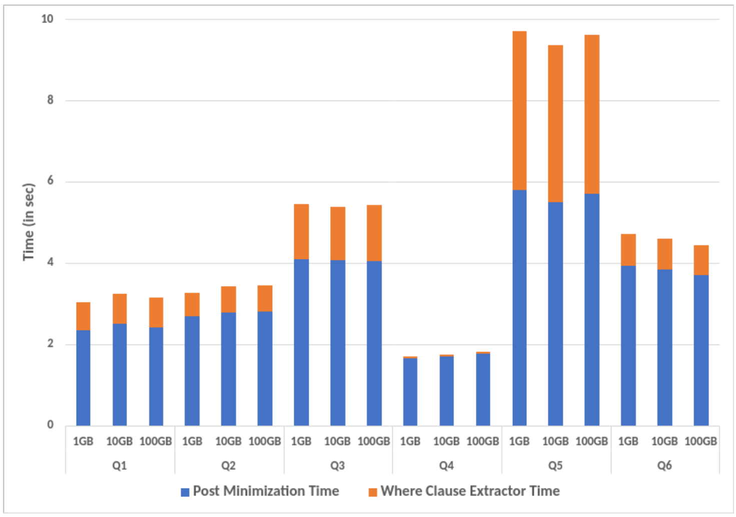


Figure 6.1: Extraction Time Comparison

6.2 Equi-Join Extraction

We have removed the *Join Predicate Extractor* module used to identify inner Equi-Joins between key columns and identifying all the Joins using modified *Where Clause Extractor* module. But if we already know that there will be only inner Equi-Join between key columns in the hidden query, in that case, which one will be better to use. UNMASQUE's Equi-Join extraction works on schema graph and result cardinality, whereas in our case, Equi-Join works on the resulting cardinality and content. So, we have created the queries with different numbers of Equi-Joins

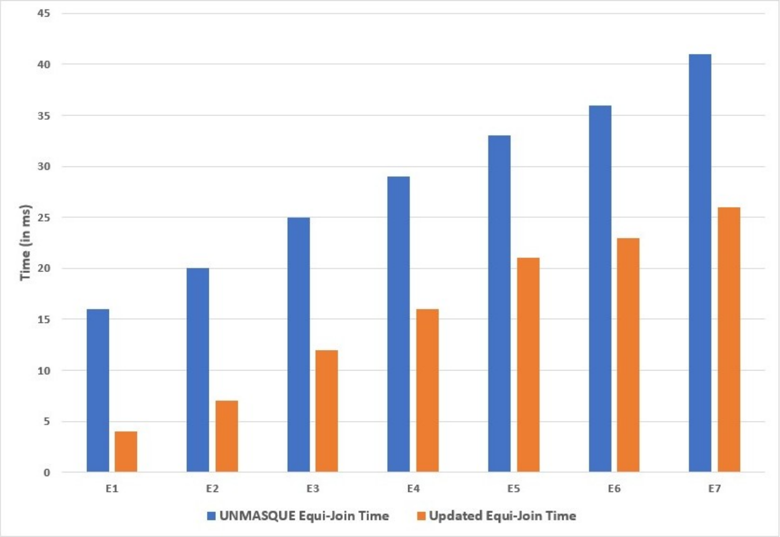


Figure 6.2: Equi-Join Extraction Time Comparison

based on TPC-H database.

The extraction time for the updated UNMASQUE has significant improvements. Also, we can use this approach to find intra-table equality predicates and inter-table equality predicates (i.e., Equi-Joins) between any pair of columns. So, we can continue with this approach in the original UMNASQUE without adding any overhead.

6.3 Extraction Time wrt Modules

For this experiment, we have analyzed the time consumption for both the modules that comes under *Where Clause Extractor* i.e., *Active Predicate Extractor* and *Dormant Predicate Extractor*.

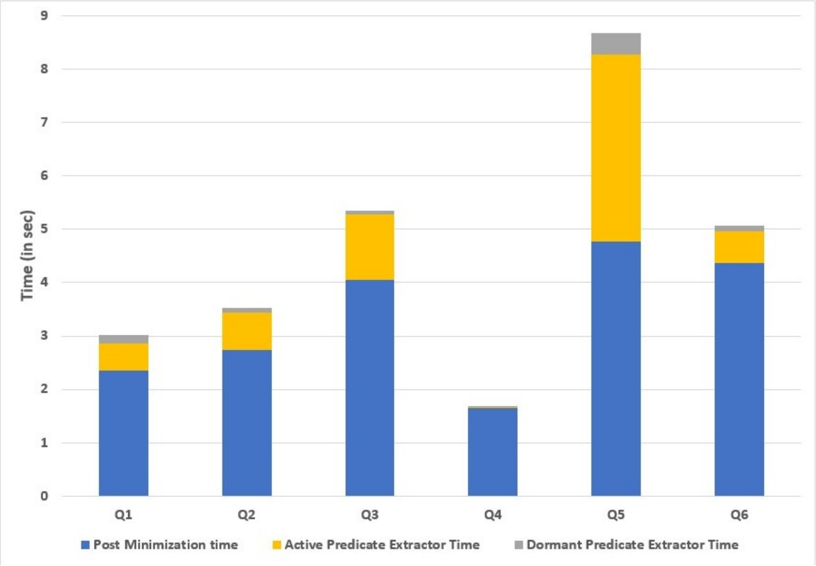


Figure 6.3: Module-wise Time Comparison

Active Predicate Extractor is consuming most of the time during the Where Clause Extraction, which is expected because there will be two Active Predicates for a single Dormant Predicate. Hence, the number of Active Predicates will be more. Also, in order to extract dormant predicates, we are converting them to active predicates.

6.4 Overhead Analysis

We have discussed the queries containing algebraic predicates, but what overhead we will be getting if the hidden queries lie under the original UNMASQUE's extractable domain. As we are finding the bounds for each columns which is same as original UNMASQUE, but we will be doing an extra check of viable candidates for Algebraic Predicates (i.e., variable bound). Here, no column turned out to be a legit contender leading to the termination of the process.

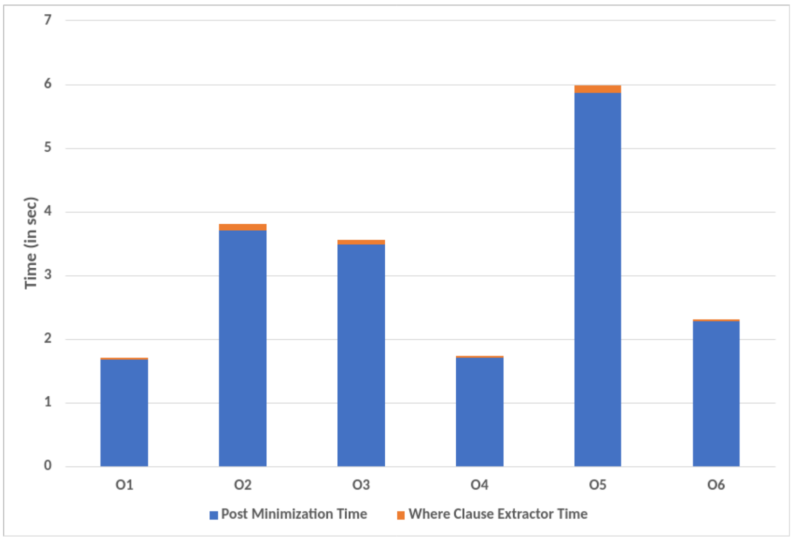


Figure 6.4: Overhead Analysis

There will be negligible overhead and can be seen from the experiments.

Chapter 7

Conclusion and Future Work

The updated UNMASQUE can extract the queries containing Algebraic Predicates, including the inner Equi-Joins and Non-Equi Joins between any pair of columns. The key concept used by the original UNMASQUE for the Filter Predicate extraction is based on result cardinality. In contrast, the updated one has extended this by analyzing the result data. This work can be used as a fundamental building block to extract multi-column predicates, more complex UDF predicates in the Where Clause, etc.

Some operators can not be extracted by UNMASQUE yet. One possible direction for future work would be to develop new ideas to extract the Outer Joins, Nested Correlation, etc.

References

[1] *PostgreSQL*. `https://www.postgresql.org/`. 3

[2] *TPC-DS*. `www.tpc.org/tpcds/`. 2

[3] *TPC-H*. `www.tpc.org/tpch/`. 2

[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. ii, 1, 8, 9, 13

[5] K. Khurana and J. Haritsa. *Opaque Query Extraction*. Technical Report. Indian Institute of Science, March 2021. `https://dsl.cds.iisc.ac.in/publications/report/TR/TR-2021-02_updated.pdf`. 9, 13

Appendix

Queries with Algebraic Predicates

Q1 :

Select l_shipmode, count(*) as count
From orders, lineitem
Where o_orderkey = l_orderkey and l_commitdate < l_receiptdate and l_shipdate < l_commitdate and l_receiptdate >= ‘1994-01-01’ and l_receiptdate < ‘1995-01-01’ and l_extendedprice $\leq$ o_totalprice and l_extendedprice $\leq$ 70000 and o_totalprice > 60000
Group By l_shipmode
Order By l_shipmode

Q2 :

Select o_orderpriority, count(*) as order_count
From orders, lineitem
Where l_orderkey = o_orderkey and o_orderdate >= ‘1993-07-01’ and o_orderdate < ‘1993-10-01’ and l_commitdate < l_receiptdate
Group By o_orderpriority
Order By o_orderpriority

Q3 :

Select l_orderkey, l_linenumber
From orders, lineitem, partsupp
Where ps_partkey = l_partkey and ps_suppkey = l_suppkey and o_orderkey = l_orderkey and l_shipdate >= o_orderdate and ps_availqty <= l_linenumber
Order By l_orderkey
Limit 10

Q4 :
Select l_shipmode
From lineitem, partsupp
Where ps_partkey = l_partkey and ps_suppkey = l_suppkey and ps_availqty = l_linenumber
Group By l_shipmode
Order By l_shipmode
Limit 5

Q5 :
Select l_orderkey, l_linenumber
From orders, lineitem, partsupp
Where o_orderkey = l_orderkey and ps_partkey = l_partkey and ps_suppkey = l_suppkey and ps_availqty = l_linenumber and l_shipdate >= o_orderdate and o_orderdate >= ‘1990-01-01’ and l_commitdate <= l_receiptdate and l_shipdate <= l_commitdate and l_receiptdate > ‘1994-01-01’
Order By l_orderkey
Limit 7

Q6 :
Select s_name, count(*) as numwait
From supplier, lineitem, orders, nation
Where s_suppkey = l_suppkey and o_orderkey = l_orderkey and o_orderstatus = ‘F’ and l_receiptdate >= l_commitdate and s_nationkey = n_nationkey
Group By s_name
Order By numwait desc
Limit 100

Equi-Join Queries

E1 :
Select *
From customer, orders
Where c_custkey = o_custkey

E2 :
Select *

From customer, orders, lineitem
Where c_custkey = o_custkey and o_orderkey = l_orderkey

E3 :
Select *
From customer, orders, lineitem, nation
Where c_custkey = o_custkey and o_orderkey = l_orderkey and n_nationkey = c_nationkey

E4 :
Select *
From customer, orders, lineitem, nation, part
Where c_custkey = o_custkey and o_orderkey = l_orderkey and n_nationkey = c_nationkey and p_partkey = l_partkey

E5 :
Select *
From customer, orders, lineitem, nation, part, region
Where c_custkey = o_custkey and o_orderkey = l_orderkey and n_nationkey = c_nationkey and p_partkey = l_partkey and r_regionkey = n_regionkey

E6 :
Select *
From customer, orders, lineitem, nation, part, region, partsupp
Where c_custkey = o_custkey and o_orderkey = l_orderkey and n_nationkey = c_nationkey and p_partkey = l_partkey and r_regionkey = n_regionkey and p_partkey = ps_partkey

E7 :
Select *
From customer, orders, lineitem, nation, part, region, partsupp, supplier
Where c_custkey = o_custkey and o_orderkey = l_orderkey and n_nationkey = c_nationkey and p_partkey = l_partkey and r_regionkey = n_regionkey and p_partkey = ps_partkey and s_suppkey = ps_suppkey

Queries w/o Algebraic Predicates

O1 :

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’ - interval ‘71 days’
Group By l_returnflag, l_linestatus
Order By l_returnflag, l_linestatus

O2 :

Select s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone, s_comment
From part, supplier, partsupp, nation, region
Where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_size = 38 and p_type like ‘%TIN’ and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = ‘MIDDLE EAST’
Order By s_acctbal desc, n_name, s_name, p_partkey
Limit 100

O3 :

Select l_orderkey, sum(l_discount) as revenue, o_orderdate, o_shippriority
From customer, orders, lineitem
Where c_mktsegment = ‘BUILDING’ and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < ‘1995-03-15’ and l_shipdate > ‘1995-03-15’
Group By l_orderkey, o_orderdate, o_shippriority
Order By revenue desc, o_orderdate, l_orderkey
Limit 10

O4 :

Select o_orderdate, o_orderpriority, count(*) as order_count
From orders
Where o_orderdate >= date ‘1997-07-01’ and o_orderdate < date ‘1997-07-01’ + interval ‘3’ month
Group By o_orderdate, o_orderpriority

Order By o_orderpriority
Limit 10

O5 :
Select n_name, sum(l_extendedprice) as revenue
From customer, orders, lineitem, supplier, nation, region
Where c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey = s_nationkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = ‘MIDDLE EAST’ and o_orderdate >= date ‘1994-01-01’ and o_orderdate < date ‘1994-01-01’ + interval ‘1’ year
Group By n_name
Order By revenue desc
Limit 100

O6 :
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 < 24
Group By l_shipmode
Limit 100