Hidden Query Extraction

Kapil Khurana Jayant Haritsa

Technical Report
TR-2020-01

Database Systems Lab
Dept. of Computational and Data Sciences
Indian Institute of Science
Bangalore 560012, India

http://dsl.cds.iisc.ac.in

Abstract

We investigate here the query reverse-engineering problem of unmasking SQL queries hidden within database applications. As a first step in addressing this challenge, we present UNMASQUE, an extraction algorithm that is capable of identifying a substantive class of complex hidden queries. A special feature of our design is that the extraction is non-invasive w.r.t. the application, examining only the results obtained from its executions on databases derived with a combination of data mutation and data generation techniques. Further, potent optimizations, such as database size reduction to a few rows, are incorporated to minimize the extraction overheads. A detailed evaluation over benchmark databases demonstrates that UNMASQUE is capable of correctly and efficiently extracting a broad spectrum of hidden queries.

1 Introduction

Over the past decade, *query reverse-engineering* (QRE) has evinced considerable interest from both the database and programming language communities (e.g. [26, 22, 20, 19, 10, 4, 1, 14, 5, 24]). The generic problem tackled in this stream of work is the following: Given a database instance $\mathcal{D}_i$ and a populated result $\mathcal{R}_i$, identify a candidate SQL query $\mathcal{Q}_c$ such that $\mathcal{Q}_c(\mathcal{D}_i) = \mathcal{R}_i$. The motivation for QRE stems from a variety of use-cases, including: (i) reconstruction of lost queries; (ii) query formulation assistance for naive SQL users; (iii) enhancement of database usability through a slate of instance-equivalent candidate queries; and (iv) explanation for unexpectedly missing tuples in the result.

Impressive progress has been made on addressing the QRE problem, with potent tools such as Talos[22], Regal[20] and Scythe[24] having been developed over the years. Notwithstanding, there are intrinsic challenges underlying the problem framework: First, the choice of candidate query is organically dependent on the specific ($\mathcal{D}_i$,$\mathcal{R}_i$) instance provided by the user, and can vary hugely based on this initial sample. As an extreme case in point, if the result has only a single row, the generated candidates are likely to be trivial queries, although the ideal answer may be an aggregation query. Second, given the inherently exponential search space of alternatives, identifying and selecting among the candidates is not easily amenable to efficient processing. Third, the precise values of filter predicates, as well as advanced SQL constructs (e.g. LIMIT, LIKE), are fundamentally impossible to deduce since the candidate query is constructed solely from the instance.

In this report, we consider a variant of the QRE problem, wherein a *ground-truth* query is additionally available, but in a hidden form that is not easily accessible. For example, the original query may be explicitly hidden in a black-box application executable. Moreover, encryption or obfuscation may have been additionally 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 arise in the context of legacy code, where the original source has been lost or misplaced over time (prominent instances of such losses are recounted in [41]), or when third-party proprietary tools are part of the workflow, or if the software has been inherited from external developers.

Formally, we introduce the hidden-query extraction (HQE) variant of QRE as follows: *Given a black-box application $\mathcal{A}$ containing a hidden SQL query $\mathcal{Q}_H$, and a database instance $\mathcal{D}_i$ on which $\mathcal{Q}_H$ produces a populated result $\mathcal{R}_i$, unmask $\mathcal{Q}_H$ to reveal the original query.*

We leverage the presence of the hidden ground-truth to deliver a variety of advantages:

- The outcome now becomes independent of the initial ($\mathcal{D}_i$,$\mathcal{R}_i$) instance.
- Since the application can be invoked repeatedly on different databases, efficient and focused mechanisms can be designed to precisely identify the hidden query.
- It allows for capturing difficult SQL constructs (we show here how LIKE and LIMIT can be extracted).
- As a collateral benefit, the unmasked query can serve as a definitive seed input to database usability tools like Talos[22] which create an array of instance-equivalent queries.
- New use-cases become feasible – for instance, a security agency may wish to ascertain offline the real intent of encrypted queries that were refused entry due to concerns about their origins.

At first glance it may appear that the existing QRE techniques could be used to provide a *seed query* for HQE, followed by refinements to precisely identify the hidden query. However, as explained later, this is not a viable approach, forcing us to design the extraction procedures from scratch. Our experience in this effort is that HQE proves to be a challenging research problem due to factors such as (a) acute dependencies between the various clauses of the hidden query, (b) possibility of schematic renaming, and (c) result compression due to aggregation functions.

```
Select    l_orderkey, sum(l_extendedprice) as revenue,
          o_orderdate, o_shippriority
From      customer, orders, lineitem
Where     c_custkey = o_custkey  and l_orderkey = o_orderkey
          and c_mktsegment = 'BUILDING'
          and o_orderdate < date '1995-03-15'
          and l_shipdate > date '1995-03-15'
group by  l_orderkey, o_orderdate, o_shippriority
order by  revenue desc, o_orderdate
limit     10;
```

(a) **Hidden Query** ($\mathcal{Q}_H$)

```
Select    l_orderkey, sum(l_extendedprice) as revenue,
          o_orderdate, o_shippriority
From      customer, lineitem, orders
Where     c_custkey = o_custkey  and l_orderkey = o_orderkey
          and c_mktsegment = 'BUILDING'
          and o_orderdate <= date '1995-03-14'
          and l_shipdate >= date '1995-03-16'
group by  l_orderkey, o_shippriority, o_orderdate
order by  revenue desc, o_orderdate asc
limit     10;
```

(b) **Extracted Query** ($\mathcal{Q}_E$)

```
Select    min(l_orderkey), sum(l_extendedprice) as revenue,
          o_orderdate, min(o_shippriority)
From      customer, orders, lineitem
Where     c_custkey = o_custkey
          and l_orderkey = o_orderkey
          and o_orderdate between '1994-11-19'
          and '1995-03-10' and c_mktsegment = 'BUILDING'
          and l_shipdate between '1995-03-20'
          and '1995-07-12'
group by  o_orderdate
```

(c) **Sample Regal Query**

Figure 1: Hidden Query Extraction Example (TPC-H Q3)

UNMASQUE Algorithm

We take a first step towards addressing the HQE problem here by presenting UNMASQUE[1], an algorithm that uses a judicious combination of *database mutation* and synthetic *database generation* to identify the hidden query $\mathcal{Q}_H$. The extraction is completely *non-invasive* wrt the application code, examining only the results obtained from its executions on carefully constructed databases. As a result, platform-independence is achieved wrt the underlying database engine.

Currently, UNMASQUE is capable of extracting a restricted but substantive class of SPJGAOL[2] queries. As an exemplar, consider the hidden query shown in Figure 1a, which features all these clauses and is based on Query 3 of the TPC-H benchmark.[3] Our extracted equivalent, $\mathcal{Q}_E$, is shown in Figure 1b, clearly capturing all *semantic* aspects of the original query. Only syntactic differences, such as a different grouping column order, remain in the extraction.

As a reference point, the candidate query produced by Regal+[20] for this scenario is shown in Figure 1c. While the query tables and joins are detected correctly, there are significant discrepancies

[1]Unified Non-invasive MAchine for Sql QUery Extraction

[2]SELECT, PROJECT, JOIN, GROUPBY, AGGREGATION, ORDER, LIMIT

[3]The only modification is that the algebraic function in the SELECT clause on l_extendedprice is curtailed to a simple aggregation.

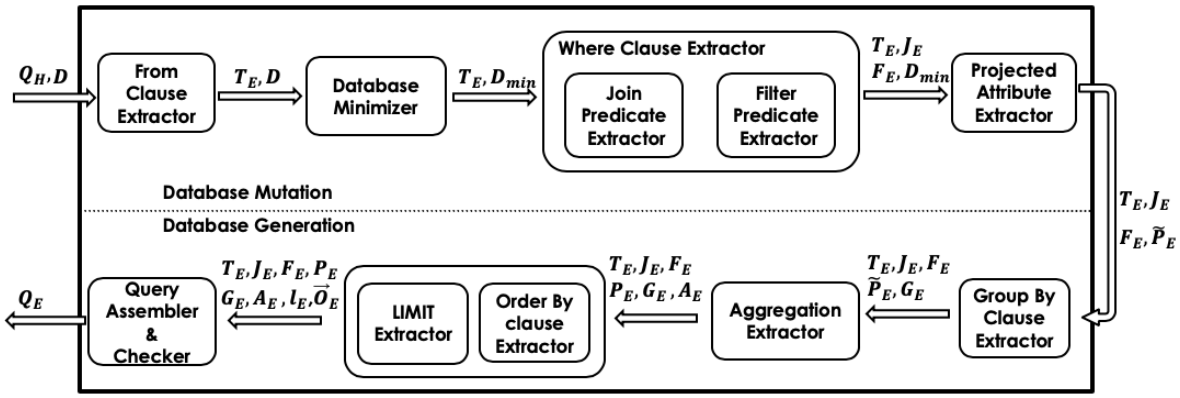

Figure 2: UNMASQUE Architecture

in the filters, grouping columns and aggregation functions. Moreover, the query is produced after removing limit, order and UDF clauses and converting character and date type columns to integers to suite their environment. Finally, producing even this limited outcome took considerable time and resources.

Extraction Workflow

UNMASQUE operates according to the pipeline shown in Figure 2, where it unmasks the hidden query elements in a structured manner. It starts with the FROM clause, continues on to the JOIN and FILTER predicates, follows up with the PROJECTION and GROUP BY+AGGREGATION columns, and concludes with the ORDER BY and LIMIT functions. The initial elements are extracted using *database mutation* strategies, whereas the subsequent ones are extracted leveraging *database generation* techniques. Further, while some of the elements are relatively easy to extract (e.g. FROM), there are others (e.g. GROUP BY) that require carefully crafted methods for unambiguous identification. The final component in the pipeline is the QUERY ASSEMBLER which puts together the different elements of $\mathcal{Q}_E$ and performs canonification to ensure a standard output format.

Extraction Efficiency

To cater to extraction efficiency concerns, UNMASQUE incorporates a variety of optimizations. In particular, it solves a conceptual problem of independent interest: *Given a database instance $\mathcal{D}$ on which a hidden query $\mathcal{Q}_H$ produces a populated result $\mathcal{R}$, identify the smallest subset $\mathcal{D}_{min}$ of $\mathcal{D}$ such that the result of $\mathcal{Q}_H$ continues to be populated.*

At first glance, it may appear that $\mathcal{D}_{min}$ can be easily obtained using well-established provenance techniques (e.g. [11]). However, due to the *hidden* nature of $\mathcal{Q}_H$, these approaches are no longer viable. Therefore, we design alternative strategies based on a combination of sampling and recursive database partitioning to achieve the minimization objective.

The database minimization is applied immediately after the FROM clause has been identified, as shown in Figure 2. And the reduction is always to the extent that the subsequent SPJ extraction is carried out on *miniscule* databases containing just a handful of rows. In an analogous fashion, the synthetic databases created for the GAOL extraction are also carefully designed to be very thinly populated. Overall, these reductions make the post-minimization processing to be essentially independent of database size.

Performance Evaluation

We have evaluated UNMASQUE's behavior on a suite of complex decision-support queries, and on imperative code sourced from blogging tools. The performance results of these experiments, conducted on a vanilla PostgreSQL platform, indicate that UNMASQUE precisely identifies the hidden queries in our workloads in a timely manner. As a case in point, the extraction of the example Q3 on a 100 GB TPCH database was completed within *10 minutes*. This performance is especially attractive considering that a native execution of Q3 takes around 5 minutes on the same platform.

Organization

The rest of the report is organized as follows: In Section 2, a precise description of the HQE problem is provided, along with the notations. The following sections – Sections 3 and 4 – present the components of the UNMASQUE pipeline, which progressively reveal different facets of the hidden query. The experimental framework and performance results are reported in Section 5. Finally, our conclusions and future research avenues are summarized in Section 6.

2 Problem Framework

We assume that an application executable object file is provided, which contains either a single SQL query or imperative logic that can be expressed in a single query. If there are multiple queries in the application, we assume that each of them is invoked with a separate function call, and not batched together, reducing to the single query scenario. This assumption is consistent with open source projects such as *Wilos* [45], which contain code segments wherein each function implements the logic of a single relational query.

If the hidden SQL query is present as-is in the executable, it can be trivially extracted using standard string extraction tools (e.g. *Strings* [34]). However, if there has been post-processing, such as *encryption* or *obfuscation*, for protecting the application logic, this option is not feasible. An alternative strategy is to re-engineer the query from the *execution plan* at the database engine. However, this knowledge is also often not accessible – for instance, the SQL Shield tool[42] blocks out plan visibility in addition to obfuscating the query. Finally, if the query has been expressed in imperative code, then neither approach is feasible for extraction.

Moving on to the database contents, there is no inherent restriction on column data types, but we assume for simplicity, the common *numeric* (int, bigint and float with fixed precision), *character* (char, varchar, text), *date* and *boolean* types. The database is freely accessible through its API, supporting all standard DML and DDL operations, including creation of a test silo in the database for extraction purposes.

2.1 Extractable Query Class

The QRE literature has primarily focused on constructing generic SPJGA queries that do not feature non-equi-joins, nesting, disjunctions or UDFs. We share some of the restrictions but have been able to extend the query extraction scope to include OL constructs. Further, we expect join graph to be a subgraph of schema graph. There are additional mild constraints on some of the constructs – for instance, the LIMIT value must be at least 3, there are no filters on key attributes – and they are mentioned in the relevant locations in the following sections. We hereafter refer to this class of supported queries

Symbol	Meaning	Symbol	Meaning(wrt query $\mathcal{Q}_E$)
$\mathcal{A}$	Application	T_E	Set of tables in query
$\mathcal{F}$	Application Executable	C_E	Set of columns in T_E
D	Initial Database	JG_E	Join graph
R	Result of $\mathcal{F}$ on D	J_E	Set of join predicates
T	Set of all tables in D	F_E	Set of filter predicates
$\mathcal{Q}_H$	Hidden Query	P_E	Set of native projections with mapped result columns
$\mathcal{Q}_E$	Extracted Query	A_E	Set of aggregations with mapped result columns
D_{min}	Reduced Database	G_E	Set of group by columns
SG	Schema Graph of database	$\widetilde{P_E}$	All projections, including aggregation columns
		$\overrightarrow{O_E}$	Sequence of ordering result columns
		l_E	limit value

Table 1: Notations

as *Extractable Query Class* (EQC). Our subsequent description of UNMASQUE on EQC uses the sample TPC-H Query 3 of the Introduction (Figure 1a) as the running example.

Further, we assume a slightly simplified framework in the subsequent description – for instance, that all keys are positive integer values – the extensions to the generic cases are provided at the end.

The notations used in our description of the extraction pipeline are summarized in Table 1. To highlight its black-box nature, the application executable is denoted by $\mathcal{F}$, while $\overrightarrow{O_E}$ has a vector symbol to indicate that the ordering columns form a *sequence*.

2.2 Overview of the Extraction Approach

At first glance, it may be surmised that existing QRE techniques can be used to seed the extraction process. However, the candidate queries generated by these techniques may differ from the original query in virtually all the constructs. Moreover, they do not provide the query constructs in a compartmentalized manner, making it infeasible to selectively pull out component clauses. Finally, their performance does not easily scale to large databases – due to these reasons, we have approached the HQE problem from first principles.

To set up the extraction process, we begin by creating a silo in the database that has the same table schema as the original user database. Subsequently, all referential integrity constraints are dropped from the silo tables, since the extraction process requires the ability to construct alternative database scenarios that may not be compatible with the existing schema. We then create the following template representation for the to-be extracted query $\mathcal{Q}_E$:

Select (P_E, A_E) **From** T_E **Where** $J_E \wedge F_E$
Group By G_E **Order By** $\overrightarrow{O_E}$ **Limit** l_E;

and sequentially identify each of the constituent elements, as per the pipeline shown in Figure 2.

The initial segment of the pipeline is based on mutations of the original/reduced database and is responsible for handling the SPJ features of the query which deliver the raw query results. The modules in this segment require targeted changes to a specific table or column while keeping the rest of the database intact.

In contrast, the second pipeline segment is based on the generation of carefully-crafted synthetic databases. It caters to the GAOL query clauses, which are based on manipulation of the raw results. The modules in this segment require generation of new data for all the query-related tables under various row-cardinality and column-value constraints. We deliberately depart from the mutation approach here since these constraints may not be satisfied by the original database instance.

We hereafter refer to these two segments as the *Mutation Pipeline* and the *Generation Pipeline*, respectively, and present them in detail in the following sections.

3 Mutation Pipeline

The SPJ core of the query, corresponding to the FROM (T_E), WHERE (F_E, J_E) and SELECT (P_E) clauses, is extracted in the Mutation Pipeline segment of UNMASQUE. Aggregation columns in the SELECT clause are only identified as projections here, and subsequently refined to aggregations in the Generation Pipeline.

3.1 From Clause

An easy way to identify whether a base table t is present in $\mathcal{Q}_H$ is the following: First, temporarily rename t to $temp$. Then, execute $\mathcal{F}$ on this mutated schema and check whether it throws an error – if yes, t is part of the query; if not, terminate the nascent query execution after a short timeout period. By doing this check iteratively over all the tables in the schema, T_E can be identified.

The above "execution-with-error" approach is very quick and easy to implement. However, it may not always be feasible since either (i) the database engine may not be designed to issue such alerts, or (ii) the application may handle the error internally and not propagate the message to the user. We have therefore designed an alternative platform-agnostic approach that is based on the following observation: Given our EQC, where only inner equi-joins are permitted, if any table in the FROM clause is empty, the query result will also be empty. So, we take each candidate table t in turn, rename it to $temp$, then create a new empty table t with the same schema as $temp$. Let this modified database be called D_{mut}. Subsequently, $\mathcal{F}$ is run on D_{mut} and the result is observed – if empty, t belongs to T_E. After that, table t is dropped and $temp$ is renamed to t – this transforms D_{mut} back to the initial database instance.

The above procedure for $Q3$ on the TPC-H schema produces

T_E = {***customer, lineitem, orders***}.

Lemma: For a query Q $\in EQC$, UNMASQUE extracts the tables in the From clause precisely.

Proof. If an empty table t is in the From clause of the query Q, joining it with any other table produces zero rows due to EQC's restriction to inner equi-joins. By induction, all further joins would produce zero rows and hence, the output would be empty. On the other hand, if t is not in the From clause, the state of its contents would not affect the result of Q. Finally, the check for each table t is on a D_{mut} instance that is identical to D except for the change in t, guaranteeing that any change in the result is due to t being empty. □

3.2 Database Minimization

For enterprise database applications, it is likely that D is huge, and therefore repeatedly executing $\mathcal{F}$ on this large database during the extraction process may take an impractically long time. To tackle this issue, before embarking on the SPJ extraction, we attempt to *minimize* the database as far as possible while maintaining a *populated result*. Specifically, we address the following **row-minimality** problem:

Given a database instance D and an executable $\mathcal{F}$ producing a populated result on D, derive a reduced database instance D_{min} from D such that removing any row of any table in T_E results in an empty result.

With this definition of D_{min}, we can prove the following strong observation:

Lemma 1: *For the EQC of Section 2.1, there always exists a D_{min} wherein each table in T_E contains only a **single** row.*

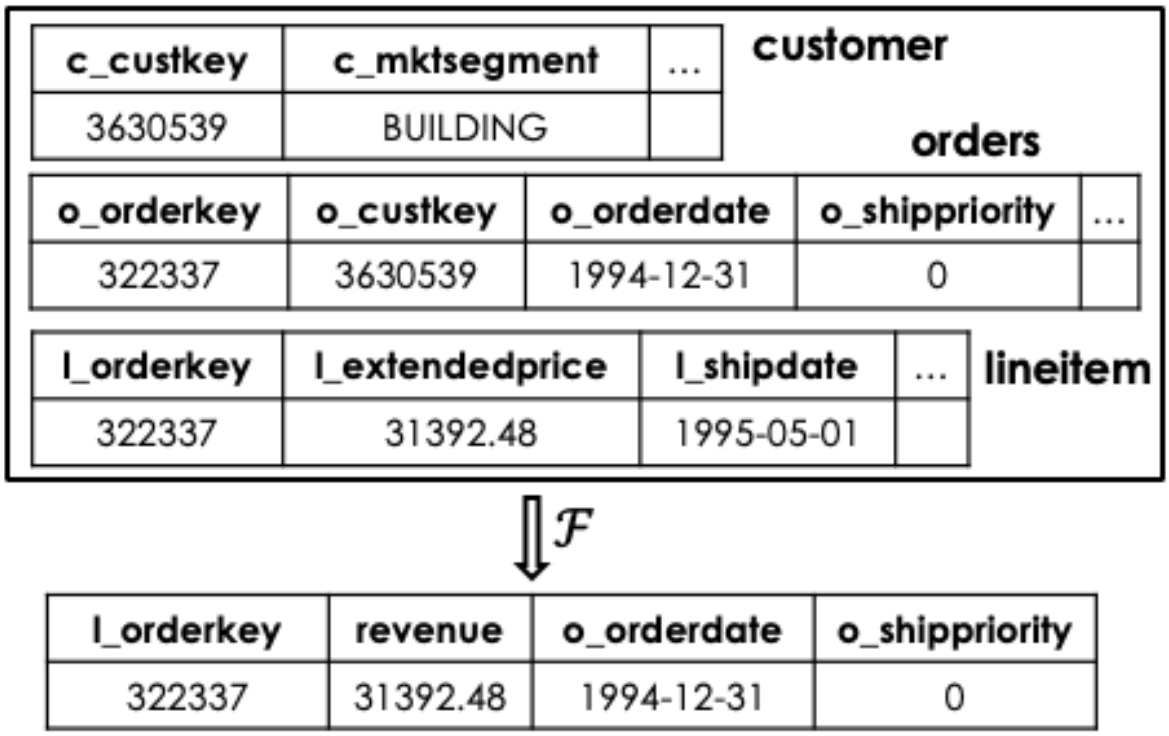

Figure 3: D^1 for Q3

Proof. Firstly, since the final result is known to be populated, the intermediate result obtained after the evaluation of the SPJ core of the query is also guaranteed to be non-empty. This is because the subsequent GAOL elements only perform computations on the intermediate result but do not add to it. Now, if we consider the *provenance* for each row r_i in the intermediate result, there will be exactly *one row* as input from each table in T_E because: (i) if there is no row from table t, r_i cannot be derived because the inner equi-join (as assumed for the query class EQC) with table t will result in an empty result; (ii) if there are $k : (k > 1)$ rows from t, $(k - 1)$ rows either do not satisfy one or more join/filter predicates and can therefore be removed from the input, or they will produce a result of more than one row since there is only a single instance of t in the query. In essence, a single-row R can be traced back to a single-row per table in D_{min}. □

We hereafter refer to this single-row D_{min} as D^1– the reduction process employed to identify this database is explained next.

Reducing D to D^1

At first glance, it might appear trivial to identify a D^1– simply pick any row from the R obtained on D and compute its provenance using the well-established techniques in the literature (e.g. [11]) – the identified source rows from T_E constitute the single-row D^1. However, these provenance techniques are predicated on *prior knowledge* of the query, making them unviable in our case where the query is hidden. Therefore, we implement the following iterative-reduction process instead: Pick a table t from T_E that contains more than one row, and divide it roughly into two halves. Run $\mathcal{F}$ on the first half, and if the result is populated, retain only this first half. Otherwise, retain only the second half, which must, by definition, have at least one result-generating row (as per Lemma 1). When eventually all the tables in T_E have been reduced to a single row by this process, we have achieved D^1.

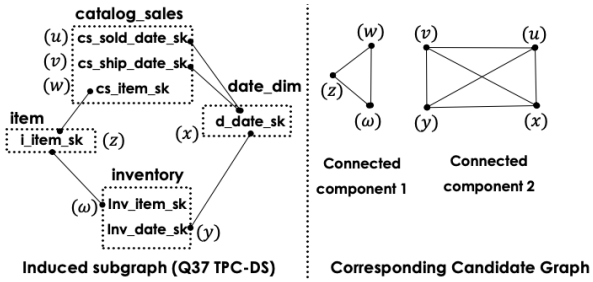

Figure 4: Induced schema graph and candidate Join-Graph for TPC-DS Q37

In principle, the tables in T_E can be progressively halved in any order. However, note that after each halving, $\mathcal{F}$ is executed to determine which half to retain, and therefore we would like to minimize the time taken by these executions. Accordingly, we choose a policy of always halving the *currently largest* table in the set. This is because this policy can be shown to require, in expectation, the least amount of data processing to reach the D^1 target.

To make the above concrete, a sample D^1 for Q3 (created from an initial 100 GB instance) is shown in Figure 3.

3.3 Join Predicates

To extract the join predicates of $\mathcal{Q}_H$, we start with the original schema graph of the database. From this graph, we create an (undirected) induced subgraph whose vertices are the key columns in T_E, and edges are the possible join edges between these columns. In the case of composite keys, each column within the key is treated as a separate node.

After that, each connected component in the subgraph is converted to a corresponding cycle graph (hereafter referred as a cycle) with same set of vertices. With slight abuse of notation, a graph with two nodes and a single edge is also referred to as a cycle. The idea behind this step is the following: Checking the presence of a connected component as one of the components of the join graph, is equivalent to checking the corresponding cycle due to the inner-equi joins in EQC. The resultant graph which is a collection of cycles is referred to as **candidate join-graph**.

If for each table t in the candidate join-graph, there is only one key column from t in the graph, and only one edge is incident on that key column, then the candidate join-graph is itself the actual joingraph, JGE, of the query. This is because for the queries in EQC, only inner equi-joins are allowed, and each table should participate in the FROM clause through a key-join. Such special case happens to be with our example query Q3.

However, the above special case may not always be applicable. To explain the procedure for arbitrary candidate join-graphs, we take another example, this time from the TPC-DS schema graph. In particular, the induced subgraph and the associated candidate join-graph for TPC-DS Q37 are shown in Figure 4. The nodes are labelled with shorthand notations to make the related figures compact and readable.

Now for each candidate cycle (say CC) in the candidate join-graph, we check its existence in the join graph. If the cycle is not present in the join graph, it is partitioned into the smaller cycles which are recursively detected for their presence in the join graph. Algorithm 1 summarizes the whole procedure. The supporting subroutines for Algorithm 1 are as follows:

Cut(C**,** e_1**,** e_2**)** Removes edges e_1 and e_2 from cycle C and returns the set of vertices in one of the newly created components.

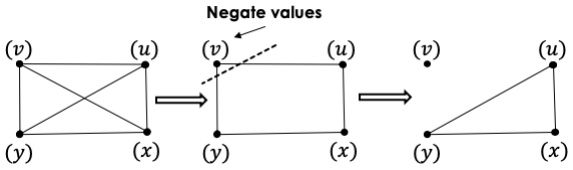

Figure 5: Checking cycle (u, v, y, x)

Negate(D^1**,** K) In D^1, values in all columns corresponding to vertices in K are replaced with the corresponding negative values.

Partition(C**,** e_1**,** e_2**)** Removes edges e_1 and e_2 from cycle C and returns two cycle graphs corresponding to the nodes in two newly created components. If any component contains a single node, then that component is not returned.

The intuition behind for loop of algorithm 1 to check the presence of a cycle (with greater than 2 nodes) is the following: if the join graph does not contain a cycle C, at least two edges of C should be absent from the join graph.

One such example is shown in Figure 5 where a populated result leads to breaking of the cycle (u, v, y, x).

Also, Algorithm 1 ensures that, in each iteration, either a cycle is removed from G or it is partitioned into two smaller cycles, thereby guaranteeing eventual termination of the algorithm.

With regard to $Q3$, the candidate graph contains only two connected components each with just a single edge. One being (l_orderkey, o_orderkey) and the other being (o_custkey, c_custkey). Thus each edge is checked for its presence and in this case, the candidate graph itself is the join graph as well. For query Q3, the added join predicates are:

J_E = {**l_orderkey = o_orderkey, o_custkey = c_custkey**}.

Lemma 2: For a hidden query $\mathcal{Q}_H \in EQC$, UNMASQUE correctly extracts JG_E, or equivalently, J_E.

Proof. It is easy see that when there is only one edge in the cycle, it will be correctly extracted as the output after removing it will be empty iff this edge is present in the join graph. For the edges that belong to bigger cycles, we prove the claim by contradiction. Consider an edge (u, v) that belongs to JG_E but UNMASQUE fails to extract it (i.e. a false negative). This implies that when the edge (u, v) is removed by value negation (with any other edge) the result continues to be populated. This is not possible if $(u, v) \in JG_E$ as one of the nodes from u and v is negated.

On the other hand, consider an edge $(u, v) \in C$ that is not part of JG_E but UNMASQUE extracts it (i.e. a false positive). This implies that when the edge (u, v) is explicitly removed along with any other edge (x, y) by value negation, the result becomes empty. As there is no other filter on key attributes and $(u, v) \notin JG_E$, every other edge in C must belong to the join graph. Now due to inner-equi joins (u, v) also belongs to the join graph as it can be inferred by other edges of cycle C, a contradiction. □

Example extracted graph along with actual join graph is shown in Figure 6. Note that, there is an extra edge *(cs_item_sk, inv_item_sk)* in the extracted graph. However, due to inner equi-joins, the two join graphs are semantically equivalent as this edge can be inferred from *(i_item_sk, inv_item_sk)* and *(cs_item_sk, i_item_sk)*.

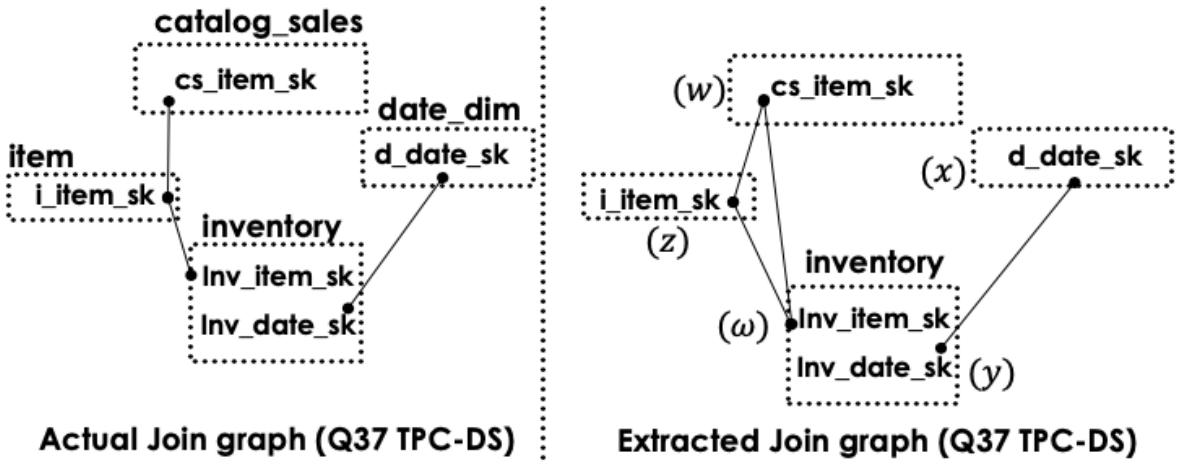

Figure 6: Actual and Extracted join Graphs

Algorithm 1: Getting Join Graph JG_E

$G \leftarrow$ Candidate Join Graph(cycles), $JG_E \leftarrow phi$
while *There is at least one cycle in G* **do**
  $C \leftarrow$ Any cycle from G
  if *C contains a single edge* (v_1, v_2) **then**
    $K \leftarrow v_1$; $D^1_{mut} \leftarrow$ Negate(D^1, K)
    If $\mathcal{F}\ (D^1_{mut}) = \phi$ **then** $JG_E \leftarrow JG_E \cup C$
    $G \leftarrow G\ /\ C$
  else
    foreach *pair of edges* $(e_1, e_2) \in C$ **do**
      K = $Cut(C, e_1, e_2)$
      $D^1_{mut} \leftarrow$ Negate(D^1, K)
      if $\mathcal{F}\ (D^1_{mut}) = \phi$ **then**
        Add e_1, and e_2 back to C
      else
        $G \leftarrow G \cup$ Partition(C, e_1, e_2)
        break //Go to the start of while loop
      end
    end
    $JG_E \leftarrow JG_E \cup C$; $G \leftarrow G\ /\ C$
  end
end

3.4 Filter Predicates

We start by assuming that all columns in C_E are potential candidates for the filter predicates F_E in $\mathcal{Q}_H$. Each of them is then checked in turn with the following procedure: First, we evaluate whether there is a nullity-related predicate on the column. If an *IS NULL* predicate is not present, we investigate whether there is an arithmetic predicate, and if yes, the filter value(s) for the predicate are identified. These steps are explained below in detail.

3.4.1 Nullity Predicates

A column value can satisfy only one of the two nullity predicates – IS NULL and IS NOT NULL. Thus, we create a mutated version of D^1 to check for the existence of one or the other, as follows: For a column $t.A$ having *NULL* in D^1, create a D^1_{mut} instance by replacing the NULL with any other value in the domain of $t.A$. If running $\mathcal{F}$ on this D^1_{mut} gives an empty result, add *"A IS NULL"* to F_E. On the other hand, for an attribute $t.A$ having any value other than *NULL* in D^1, create a D^1_{mut} instance by replacing the value of $t.A$ with *NULL*. If running $\mathcal{F}$ on this D^1_{mut} gives an empty result, add *"A IS NOT NULL"* to F_E.

3.4.2 Numeric Predicates

For ease of presentation, we start by explaining the basic idea for *integer* columns. Let $[i_{min}, i_{max}]$ be the value range of column A's integer domain. We start by assuming that there is 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 $i_{min} \leq A \leq 24$.

Now, to check if there is a filter predicate on column A, we first create a D^1_{mut} instance by replacing the value of A with i_{min} in D^1, then run $\mathcal{F}$ and get the result – call it R_1. We get another result – call it R_2 – 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.

Case	$R_1 = \phi$	$R_2 = \phi$	**Predicate Type**	**Action Required**
1	No	No	$i_{min} \leq A \leq i_{max}$	No Action
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

Table 2: Filter Predicate Cases

These cases hold because there is no other filter source (e.g. HAVING clause) in EQC. Now, if the match is with Case 2 (resp. 3), we can use a binary-search-based approach over $(i_{min}, i_{max}]$ (resp. $[i_{min}, i_{max})$), to identify the specific value of l (resp. r). However, a better strategy would be to utilize the value of A already present in D^1– if this value is a, the binary search needs to be done only over $(i_{min}, a]$ for l and over $[a, i_{max})$ for r. After this search completes, the associated predicate is added to F_E. Finally, it is trivial to see that Case 4 is a combination of Cases 2 and 3, and can therefore be handled in a similar manner.

We can easily extend the integer approach to *float* data types operating with *fixed precision*, as follows: First find the *integral* bounds as per above, and then the *fractional* bounds with a second binary search. For example, with l_i and r_i as the integral bounds identified in the first step, and assuming a precision of 2, we search l in $((l_i - 1).00, l_i.00]$ and r in $[r_i.00, r_i.99)$ in the second step.

Date Columns Extracting predicates on *date* columns is identical to that of integers, with the minimum and maximum expressible dates in the database engine serving as the initial range, and days as the difference unit. For example, after identifying filter of type $A \leq r$ on o_orderdate, we apply binary search strategy in range [*'1994-12-31', r*], where '1994-12-31' is the value of o_orderdate in D^1 (Figure 3), and r is the greatest allowed date value in the database engine (for PostgreSQL, $r = 5874897AD$). Note that the same strategy can be applied to other *datetime* type columns with the corresponding change in the resolution of values.

Boolean Columns With a single row, a boolean column can have only one of True or False values. Therefore, to identify a filter on boolean column $t.A$, we create a D^1_{mut} by replacing its value in D^1 with True (resp. False) if the current value in D^1 is False (resp. True) and get the result. If the result is empty, add *"A* = False*"* (resp. *"A* = True*"*) to F_E.

3.4.3 Character Columns

The extraction procedure for character columns is more complex because strings can be of varying length and the filters may contain wildcard characters ('_' and '%'). To check for the existence of a filter predicate, we create two different D^1_{mut} instances by replacing the value of A initially with an empty character and then with a single character – say *'a'*. The executable $\mathcal{F}$ is invoked on both these instances and the result characteristics are noted.

We conclude that there is a filter predicate in operation iff the result is empty in one or both cases. To prove the *if* part, it is easy to see that, given the absence of other filter sources in EQC, if the result is empty in either of the cases, there must be some filter criteria on A. For the *only if* part, the result will be populated for both the cases in only one extreme scenario – *A like '%'*, equivalent to *no* filter on A.

After confirming the existence of a filter predicate on A, we extract the specific predicate in two steps. Before getting into the details, we define a term called *Minimal Qualifying String (MQS)*. Given a character/string expression *val*, its MQS is the string obtained by removing all occurrences of '%' from *val*. For example, "UP_" is the MQS for *"%UP_%"*. Note that each character of MQS, with the exception of wildcard '_', must be present in the data string to satisfy the filter predicate.

Given the above formulation, the first step is to identify *MQS* using the actual value of A in D^1, denoted as the representative string, or *rep_str*. The basic idea is to loop through all the characters of *rep_str* and determine whether it is present as an intrinsic character of the MQS or invoked through the wildcards ('_' or '%'). This is achieved by replacing, in turn, each character of *rep_str* in D^1 with some other character, executing $\mathcal{F}$ on this mutated database, and checking whether the result is empty – if yes, the replaced character is part of MQS; if no, this character was invoked through wildcards. In this case, further action is taken to identify the correct wildcard character. The formal procedure is detailed in Algorithm 2. Note that in case the character in *rep_str* occurs more than once without any intrinsic character in between, and only one of them is part of MQS, our procedure puts the rightmost character in MQS.

Lemma: For a query in EQC, Algorithm 2 correctly identifies MQS for a filter predicate on character attribute.

Proof. The correctness of the algorithm 2 can be established using contradiction for each of the possible failed cases. For example, let us say a character 'a' belonged to *MQS* but the procedure fails to identify it. This means that after removing 'a' from *rep_str*, the result is still non-empty (the filter condition was satisfied). This is possible when 'a' occurs more than once in *rep_str* and there is at least one occurrence which is part of the replacement for wildcard '%'. However, the procedure will keep removing 'a' until

Algorithm 2: Identifying Minimal Qualifying String *MQS*

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

there is no occurrence left which is part of replacement for wildcard '%'. After that, removing 'a' will lead the corresponding filter predicate to fail. If this is not the case, 'a' is not present in the *MQS*, a contradiction. Similarly, the correctness for other cases can be proved.

□

After obtaining the MQS, we need to find the locations (if any) in the string where '%' is to be placed to get the actual filter value. This is achieved with the following simple linear procedure: For each pair of consecutive characters in MQS, we insert a random character that is different from both these characters. A populated result for $\mathcal{F}$ on this mutated database instance indicates the existence of '%' between the two characters. The inserted character is removed after each iteration and we start with the initial MQS for each successive pair of consecutive characters. This makes sure that we correctly identify the locations of '%' without exceeding the character length limit for A. For the example query Q_3, the predicate value for c_mktsegment turns out to be the MQS itself, namely *'BUILDING'*.

At last, if *both* the nullity predicate *IS NOT NULL* and another filter predicate have been identified on a column $t.A$, the nullity-related predicate is extraneous and hence removed from F_E. Overall, for query Q_3, we identify the following filter predicates:

F_E = { **o_orderdate** $\leq$ **date '1995-03-14',**
l_shipdate $\geq$ **date '1995-03-16',**
c_mktsegment = 'BUILDING'}

3.5 Projections

It is a common practice to assign a user-friendly name to the result columns using the AS construct. In such a case, we need to map the result column to the corresponding projected column from the database. As mentioned earlier, the PROJECTION EXTRACTOR module of UNMASQUE extracts the set of all projected columns as native projections $\widetilde{P_E}$, some of which are subsequently refined to aggregations (A_E) in Generation Pipeline. The projected columns are extracted in two steps: (i) Candidate Identification, and (ii) Pruning.

In the first step, UNMASQUE identifies the candidate set of projected columns for each result column. This is achieved by comparing the value in the result column to the values in D^1. If the result value matches multiple columns, the correct single match is identified in the pruning step. In the pruning step, a D^1_{mut} instance is created by mutating the values of candidate columns in D^1 such that each candidate column gets a unique value while satisfying the filter and join predicate(s). We begin by assigning values to the columns with filter predicates on them. This operation will be different for different data types, and we explain the procedure for numeric data types here. For all the numeric columns with filter predicates, we first compute the set of non-overlapping ranges from the set of allowed ranges in each such column. Then, for each column in this class, we assign a random value from the corresponding unique range. The remaining columns are assigned values by traversing the sorted set of all assigned values and picking a value that has not yet been utilized. Similar methods can be used to assign values to columns of other types as well. For example, for character columns with filter predicates, the wildcard characters are used to generate unique values. Then, $\mathcal{F}$ is executed and the projected column is identified as the one whose value matches the result column value; this mapping is added to $\widetilde{P_E}$. A problem, however, is that it may not always be possible to mutate D^1 in the above manner. For example, let the candidate columns be $\{A, B, C\}$ with filter constraints $\{A = 2, B = 3, 2 \leq C \leq 3\}$. In such cases, the candidates cannot be removed in one go, but instead have to be iteratively pruned one by one. Further, if multiple candidates continue to remain due to matching filter or join predicates, any one of the candidates can be selected. For example, if the filter constraints are $\{A = 3, B = 3, C = 3\}$, then any of A, B, C can be treated as a projected column. Finally, when the result column represents count(*), it will not have a matching column. Such columns are added to $\widetilde{P_E}$ with no mapping and are subsequently characterized by the Aggregation 4.2 module.
Applying the above procedure to query Q3, we get:
$\widetilde{P_E}$ = {**"l_orderkey: l_orderkey", "revenue:l_extendedprice",**
"o_orderdate: o_orderdate", "o_shippriority:o_shippriority"}.

4 Generation Pipeline

The GAOL part of the query, corresponding to the GROUP BY (G_E), AGGREGATION (A_E), ORDER BY ($\overrightarrow{O_E}$) and LIMIT (l_E) clauses, is extracted in the Generation Pipeline segment of UNMASQUE. Here, synthetically generated miniscule databases are used for all the extractions, as described in the remainder of this section.

4.1 Group By Columns

For each column $t.A$ in C_E (the set of columns in T_E), we generate a database instance D_{gen} and analyze $\mathcal{F}$ (D_{gen}) for the existence of $t.A$ in the GROUP BY clause.

Assume for the moment that we have generated a D_{gen} such that the *intermediate* result produced by the SPJ part of $\mathcal{Q}_H$ contains **3** rows satisfying the following condition: $t.A$ has a common value in exactly two rows, while all other columns have the same value in all three rows. Now, if the *final* result contains **2** rows, it means that this grouping is only due to the two different values in $t.A$, making it part of G_E. This approach of such intermediate result generation is similar to those presented in [16, 23]

One may argue the need for minimum 3 rows in the intermediate result for such determination. Clearly, by generating a single row in the intermediate result, no determination can be made about grouping on the final result as no groups will be created. Now let us say, we generate a database instance such that the intermediate result produced by SPJ part of $\mathcal{Q}_H$ has two rows. To check if an attribute A belongs to *group by* clause, we need to generate a database satisfying the following condition: A has same value in both the rows and each of the other columns has different values for both the rows. This approach is feasible if only one column is there in the *group by* clause. In case of multiple columns in *group by* clause, the columns already identified in *group by* clause may be assigned values in two ways. If we assign two different values to such attributes, even if there is grouping on A, the rows will not be merged due to other grouping column with two different values. On the other hand, if we assign same values to such attributes, even if there is no grouping on A, the rows will be merged due to other grouping column with same values. However, with three rows, those other attributes being assigned a common value, do not affect the grouping. So we need at least three rows to be present in the intermediate result. Now, if generate a 2 row database satisfying the following condition: A has different values in both the rows and each of the other columns has same value for both the rows, then presence of two rows in final output may indicate either grouping on A or no grouping at all. This approach, however can be used with some additional checks. The reason for using 3 row approach is that, in the first iteration, we can identify all the three case, (i) Group by on A, (ii) No group by on A and (iii) No group by at all in the first iteration itself.

We do not explicitly check for columns with equality filter predicates. The reason being: when used with any other column that can take two different values, these columns have no affect on grouping and when used otherwise, the query is equivalent to an SPJA query which we handle separately at last.

Generating D_{gen}

We now explain how to produce the desired D_{gen} for checking the group-by membership of a generic column $t.A$. In our description, assigning $(p, q, r, ...)$ to $t.A$ means assigning value p in the first row, q in the second row, r in the third and so on. The database generation is performed differently for the following two disjoint cases:

(Case 1) $t.A \notin JG_E$ In this case, 3 rows are generated for table t and only one row in each of the other tables in T_E. For column $t.A$, any two different values p and q that satisfy all associated filter predicates are assigned. If no filter exists, any two values from $t.A$'s domain are taken (e.g. $p = 1$ and $q = 2$ for numeric). After that, we assign (p, p, q) to $t.A$.

For all other columns in t, such as $t.X$, a single value r that satisfies its associated filter predicates (if any) is selected, and (r, r, r) is assigned to $t.X$. If there is no filter, any value from its domain (e.g. $r = 1$ for numeric) is assigned. Finally, if $t.X \in JG_E$, a fixed value of $r = 1$ is assigned (consistent with the assumption of integral keys). A similar assignment policy is used for all columns belonging to the remaining tables t' in T_E.

An example D_{gen} for checking the presence of o_orderdate in G_E is shown in Figure 7. Here, the ORDERS table features 3 rows with p = '1995-03-13' and q = '1995-03-14', while the remaining tables,

LINEITEM and CUSTOMER, have single rows.

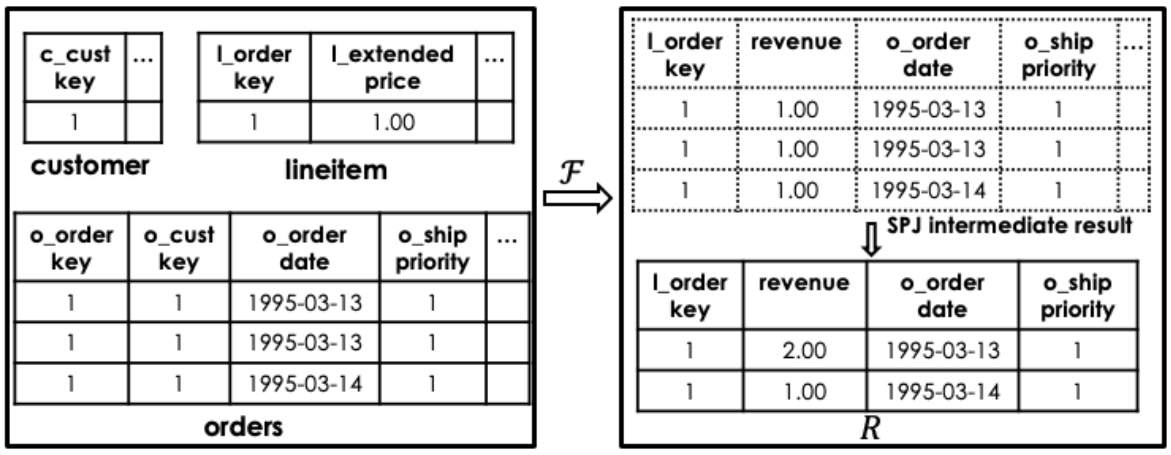

Figure 7: D_{gen} for Grouping on o_orderdate (Q3)

(Case 2) $t.A \in JG_E$ Firstly, the assignment of values to $t.A$ is similar to the policy followed above for Case 1, the only difference being that p and q are assigned fixed values of 1 and 2, respectively.

Secondly, for all tables t' having a column $t'.B$ such that there is a path between $t.A$ and $t'.B$ in JG_E, *two rows* are generated with all such $t'.B$ being assigned fixed values (1,2). All other columns are assigned values just like for $t'.X$ in Case 1, except that the assignment is now duplicated across the two rows.

Finally, for all other tables, t'', a *single row* is generated in exactly the same manner as that for $t'.X$ in Case 1.

An example D_{gen} for checking the presence of l_orderkey in G_E is shown in Figure 8. Here, there are 3 rows for LINEITEM, 2 rows for ORDERS and 1 row for CUSTOMER.

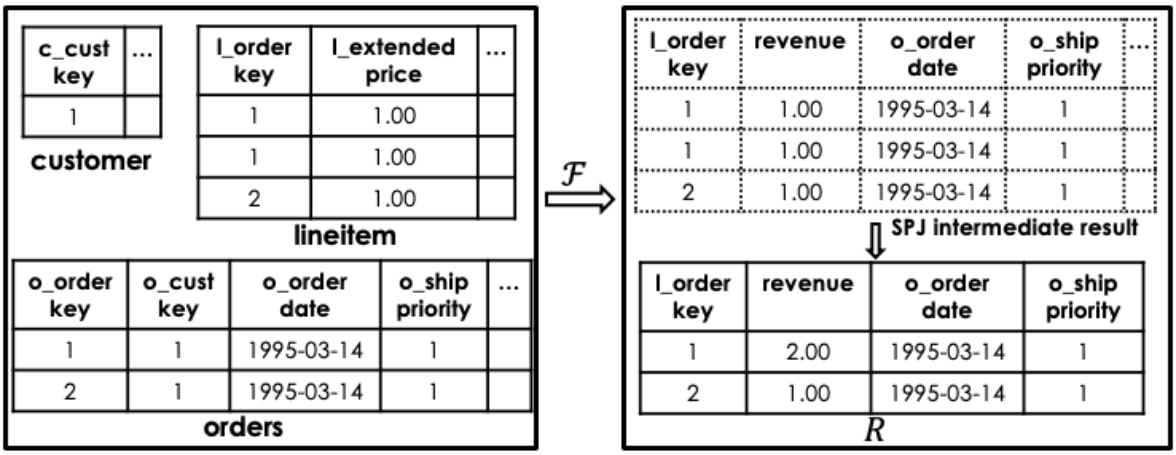

Figure 8: D_{gen} for Grouping on l_orderkey (Q3)

It is straightforward to see by inspection that, with our assumption of key-based inner equi-joins, the above data generation procedure results in ensuring the desired conditions for the intermediate SPJ result. Namely, that it will contain 3 rows with all columns having the same value across the 3 rows except for the attribute under test which has two values across these rows.

After all attributes have been processed in the above manner, if G_E turns out to be empty, we create a D_{gen} with each table having *two rows*, each column in the join-graph assigned fixed values $(1, 2)$, and any two different values to all other columns while satisfying all filter predicates. Then, $\mathcal{F}$ is run on this D_{gen}, and if the result contains just one row, we can conclude that the query has an *ungrouped* aggregation.

Overall, the above procedure produces for Q3:

G_E = {**l_orderkey, o_shippriority, o_orderdate**}.

4.2 Aggregation Functions

We explain here the procedure for identifying aggregations (*min(), max(), count(), sum(), avg()*) on numeric attributes. Similar methods can be used for textual/date attributes as well. Also, for ease of presentation, we first assume that there is no aggregation with DISTINCT keyword – such cases are explained in this section at last.

To identify an aggregation function on $t.A$ (where A is a projected attribute that is not in G_E), our objective is to generate a database D_{gen} such that the *final* result cardinality is **1**, and each of the five possible aggregation functions on $t.A$ results in a *unique value*, thereby allowing for correct identification of the specific function. We call this the "target result".

Now, for $t.A$, as we want *min()* and *max()* to be different, we need at least two different values. To ensure unique values for all the aggregations in the final output, we employ the following approach. Consider a pair of integers a and b such that $a \neq 0$ and $a \neq b$, and let $k + 1$ be the number of rows in the *intermediate* result produced by the SPJ part of the query. Further, in this intermediate result, let $t.A = a$ in k rows and $t.A = b$ in the remaining row, and k satisfy the following constraints:

$$k \notin \left\{ 0, a-1, b-1, \frac{a-b}{a}, \frac{1-b}{a-1}, \frac{(a-2) \pm \sqrt{(a-2)^2 - 4(1-b)}}{2} \right\} \tag{1}$$

These constraints on k have been derived by computing *pairwise equivalences* of the five aggregation functions, and forbidding all the k values that result in equality. Now, additionally if we ensure that the G_E attributes are assigned common values in all the rows, the result of $\mathcal{F}$ will be the target result.

The reason that the target result is produced is (i) the result cardinality is 1 since there is a common set of values for the G_E attributes, and (ii) the constraints on k ensure unique aggregated output of all the aggregated functions for $t.A$. Note that if there is a filter predicate with equality operator on $t.A$, we have to take a common value which satisfies the corresponding filter predicate. In such a case, $a = b$ and $k \notin \{0, (a\text{-}1)\}$. Here, multiple aggregations on $t.A$ may be equivalent (e.g. *min(), max(), avg()*), so we take any of the equivalent aggregation functions.

Generating D_{gen}

The data generation process to obtain the above intermediate result is similar to the D_{gen} generation of GROUP BY (explained in Section 4.1), with the following changes:

- $k + 1$ rows are generated for table t, with $t.A$ assigned value a in k rows and value b in the remaining row.
- With respect to Case 2 ($t.A \in JG_E$) in Section 4.1, the assignments of fixed values $1, 2$ are replaced with values a, b.

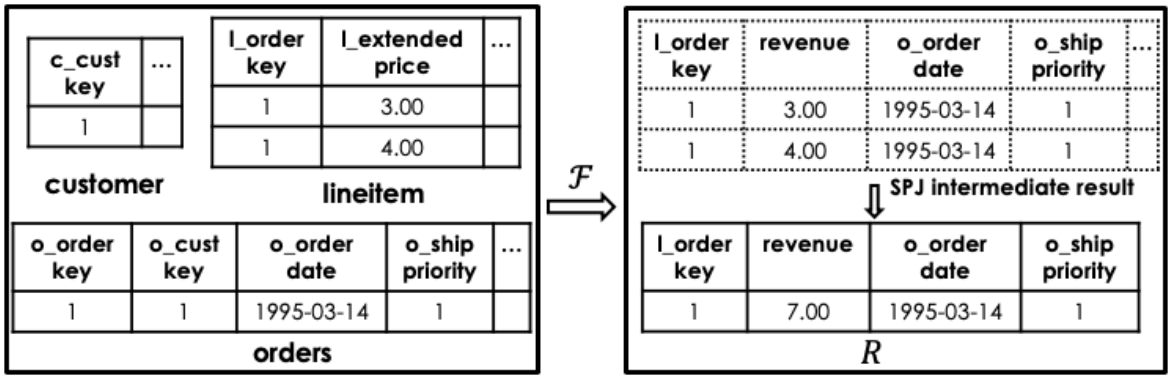

Figure 9: D_{gen} for Aggregation on l_extendedprice (Q3)

To choose a and b, any two values that are consistent with the filter predicates (if any) on $t.A$ can be chosen. Further, the least positive integer satisfying Equation 1 is chosen as k. A sample D_{gen} to check for aggregation on l_extendedprice is shown in Figure 9. As there is no filter on this attribute, we set $a = 3$ and $b = 4$ since these are the *lowest* positive values for which $k = 1$ is feasible.

After getting D_{gen}, we run $\mathcal{F}$ and the aggregation is identified by matching the result column value (corresponding to $t.A$) with the corresponding unique values for the five aggregations. The identified aggregation along with the mapping to the corresponding result column is added to A_E.

At last, entries corresponding to all the aggregated attributes are removed from $\widetilde{P_E}$ and inserted in A_E. Further, if there remains an unmapped output column in $\widetilde{P_E}$, it is removed and $count(*)$ is added to A_E. Whatever remains in $\widetilde{P_E}$ now constitutes the native (i.e. unaggregated) P_E.

With the above procedure, we finally obtain for Q3:

A_E = {$sum(l\_extendedprice)$**:revenue**}

P_E = {**"l_orderkey:l_orderkey",**
"o_orderdate:o_orderdate",
"o_shippriority:o_shippriority"}

Extension to *DISTINCT* keyword

In case the aggregation can be present with *DISTINCT* keyword as well, the following cases may happen as a result of identifying aggregation (without *distinct*) using above method:

Case1 - min() or max() aggregation is identified: In such a case, no action is required as *min()* or *max()* produces exactly same result with/without unique.

Case2 - No aggregation is identified: In such a case, the aggregation on $t.A$ is one of *sum(DISTINCT A), avg(DISTINCT A)* or *count(DISTINCT A)*. To identify the correct aggregation, we generate the D_{gen} with $t.A$ having values (p, p, q) such that $p \neq q$ and $(p + q) \notin 2, 4$ to get value for all three aggregated results unique.

Case3 - Aggregation other than min() or max() is identified: In such a case, the possible actual aggregations on A are *sum(DISTINCT A), avg(DISTINCT A), count(DISTINCT A)* or the one identified without distinct. In such a case, we generate databases to prune out this list one by one. for example, let us say that *sum (A)* is the identified projection. To prune out one of *sum(A)* and *sum(DISTINCT A)*, we generate a D_{gen} instance with k = 2 and $a \neq 0$. Similarly, other candidates can be pruned out as

well. Note that in case of equivalent aggregations, anyone can be chosen.

Extension to non-Numeric Attributes

In case of non-numeric attribute A, we need to find existence of min() or max() only. In such a case, we take k = 1 and take two different values a and b from the domain of A. The rest of the procedure remains same.

4.3 Order By

We now move on to identifying the columns present in $\overrightarrow{\mathcal{O}_E}$. As mentioned in Section 2, a special feature of ORDER BY is that it is a sequence, and not a set unlike all the other clauses in the query.

A basic difficulty here is that the result of a query can be in a particular order either due to: (i) explicit ORDER BY clause in the query or (ii) a particular plan choice (e.g. Index-based access or Sort-Merge join). Given our black-box environment, it is *fundamentally infeasible* to differentiate the two cases. However, as described below, our identification approach is such that: (a) it is unlikely to have plan-induced orderings, and (b) even if there are such extraneous orderings, the query semantics will not be altered.

For simplicity, we first assume here that $count() \notin A_E$ and that no aggregated attribute features in an equality predicate – the procedure to handle these cases is described in the section at last.

Order Extraction

We start with a candidate list comprised of the projected attributes in $P_E \cup A_E$. From this list, the attributes in $\overrightarrow{\mathcal{O}_E}$ are extracted sequentially starting from the leftmost index. The process stops when either (i) all candidates have been included, or (ii) all functionally-independent attributes of G_E have been included in $\overrightarrow{\mathcal{O}_E}$, or (iii) no sort order can be identified for the current index position.

To check for the existence of a column $t.A$ (or its corresponding AS-renamed output column) at position i, we create a pair of 2-row database instances – D^2_{same} and D^2_{rev}. In the former, the sort-order of $t.A$ is the *same* as that of all the other projected attributes, whereas in the latter, the sort-order of $t.A$ alone is *reversed* with respect to the other projected attributes. An example instance of this database pair is shown for revenue (corresponding to $sum(l\_extendedprice)$) in Figure 10.

We use the following procedure to create D^2_{same}: Firstly, for column $t.A$, assign a pair of values p and q such that $p < q$, and they satisfy the filter predicate (if any) on $t.A$. If $t.A \in JG_E$, assign fixed values $p = 1$ and $q = 2$. Then, identify all attributes $B \in JG_E$ that are already present in $\overrightarrow{\mathcal{O}_E}$. All such columns, as well as columns corresponding to their connected components in JG_E, are assigned values $(1, 1)$. The data generation for the remaining columns of all the tables is as follows: (i) Each column with a path to $t.A$ in JG_E is assigned exactly the same values that are present in $t.A$. (ii) Attributes currently present in $\overrightarrow{\mathcal{O}_E}$ are assigned a single value r satisfying their filter predicate (if any). (iii) Attributes in equality filter predicates are assigned a single value r satisfying the equality condition. (iii) For all other attributes, two values r and s are assigned such that $r < s$ and both r and s satisfy the associated filter predicate (if any).

The procedure for creating D^2_{rev} is the same as above except that (q, p) is assigned to $t.A$ so as to reverse the order.

Database construction in the above manner ensures both the rows form individual groups, so aggregated columns can be effectively treated as projections (except for $count()$, which requires a different mechanism, explained at last).

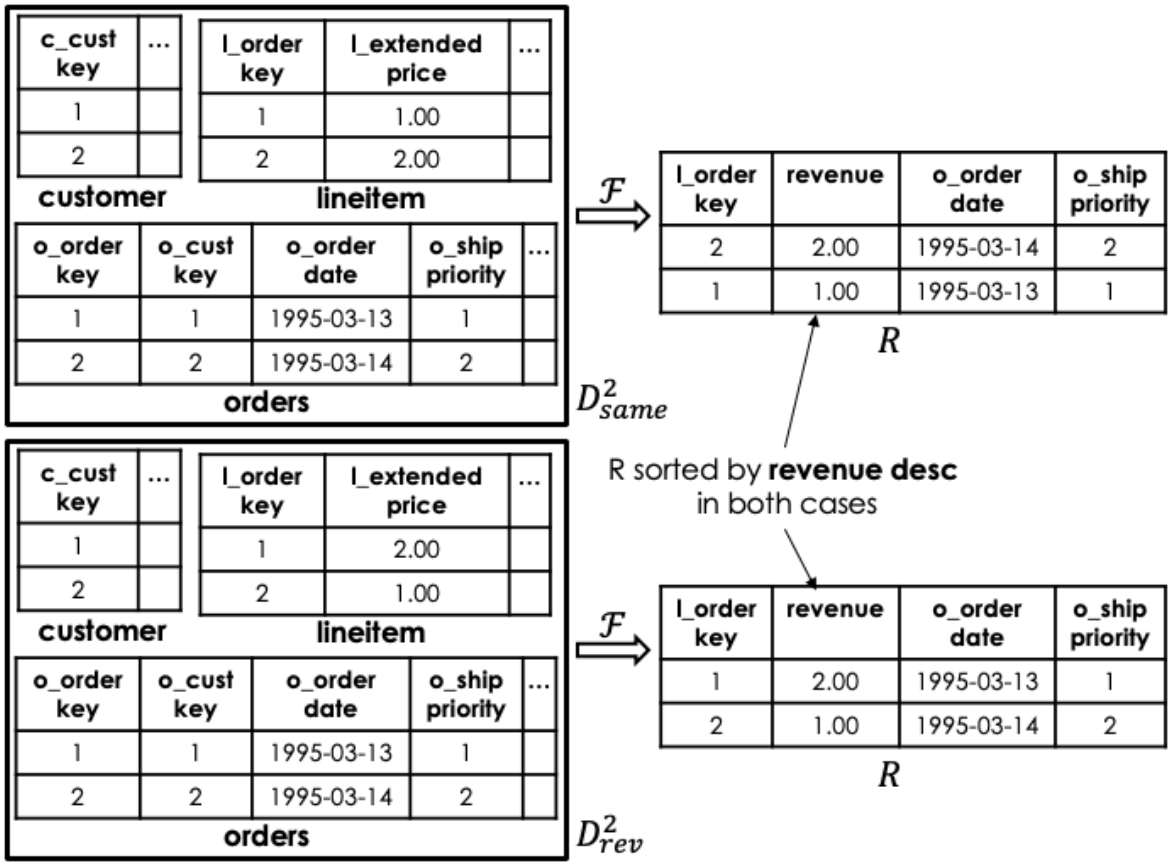

Figure 10: D^2_{same} and D^2_{rev} for Ordering on revenue (Q3)

Thus, even if $t.A$ is not a projected attribute, we can derive its ordering in the result by the ordering of the other projected attributes. After generating D^2_{same} and D^2_{rev}, we run $\mathcal{Q}_H$ for both the instances and analyze the results. If the values in $t.A$ are sorted in the same order for both the results, $t.A$ along with its associated order, is added to $\overrightarrow{\mathcal{O}_E}$ at position i. The correctness for this step is explained as follows:

Lemma 4: With the above procedure, if $t.A$ is not the rightful column at position i in $\overrightarrow{\mathcal{O}_E}$, and another attribute B is actually the correct choice, then the values in $t.A$ will not be sorted in the same order in the two results.

Proof. Firstly, as each column in the existing identified $\overrightarrow{\mathcal{O}_E}$ is assigned the same value in both the rows, they have no effect on the ordering induced by other attributes. Now, let us say that the next attribute in $\overrightarrow{\mathcal{O}_E}$ is B (asc) but UNMASQUE extracts $t.A$. Now in the result corresponding to D^2_{same}, the values in $t.A$ will also be sorted in ascending order. But in the result corresponding to D^2_{rev}, the values in A will be sorted in descending order (due to ascending order on B), a contradiction. □

With the above procedure, we finally obtain for Q3:

$\overrightarrow{\mathcal{O}_E}$ = {**revenue desc, o_orderdate asc**}

A closing note on the potential for spurious columns appearing in G_E due to plan-induced ordering: Since D^2_{same} and D^2_{rev} are extremely small in size, it is unlikely that the database engine will choose a plan with sort-based operators – for instance, it would be reasonable to expect a sequential scan rather than index access, and nested-loops join rather than sort-merge. In our experiments, we explicitly verified that this was indeed the case.

Extension1: *count(*)* $\in A_E$

In the case when *count(*)* $\in A_E$, the two rows in each of the tables is not enough as the count() value for both the groups will be one. In such case, we need an intermediate result (on which grouping will be applied) with 3 rows such that two rows form one group and the third row forms another group. Also, the values in the rows should be according to the order desired after grouping of the intermediate result. So the data generation process is as follows:

To generate data for D^2_{rev}, we first choose a table t with at least one attribute in *group by* clause that can take two different values. For each attribute $t.X$ in t, we take two different values (p and q) such that $p < q$ and assign values (p, p, q) to it. In case the attribute is a key attribute we take fixed values $p = 1$ and $q = 2$. In other cases, we take p and q satisfying the corresponding filter predicates (if any). For all the other tables t', we generate two rows with each attribute having two different values (p and q) such that $p < q$. In case of key attributes, take $p = 1 and q = 2$. In other cases, take p and q satisfying the corresponding filter predicates (if any). Note that in the above procedure, if we encounter an attribute with equality filter predicate, we take $p = q = val$ where val satisfies the corresponding filter predicate.

Data generation for D^2_{same} is similar as for D^2_{rev} with the only change being the values of p and q are now swapped. The further procedure of running $\mathcal{F}$ and analyzing the results is the same as explained in *order extraction* part of the section. A sample D^2_{same} and D^2_{rev} database instance for a hypothetical scenario where revenue is replaced by count(*) is shown in Figure 11.

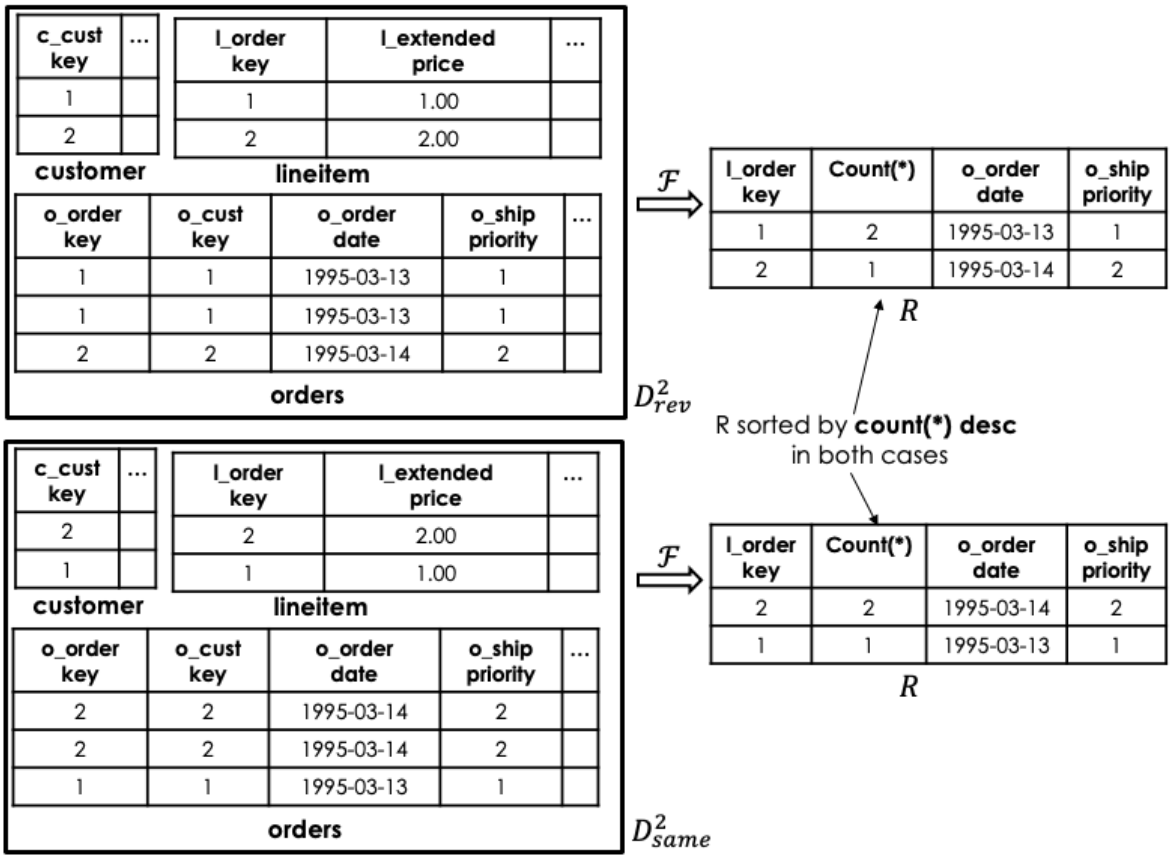

Figure 11: D^2_{same} and D^2_{rev} for Ordering on count(*) (Hypothetical scenario:Q3)

Lastly, in case *count(DISTINCT* $t.A$*)* $\in A_E$, the data generation process is the similar with the change that A is assigned values (p, q, p) in both the cases.

Extension2: $t.A : (\text{“}t.A = val\text{”} \in F_E \wedge (agg\_func(t.A) \in A_E)$

In case there is *min(), max()* or *avg()* aggregation on A, the attribute can be treated as natively projected attribute because each group in the output will have exactly the same value for A. Now, if *sum(*$t.A$*)* $\in A_E$, the data generation process is same as in Extension 1. Also note that, the aggregation case with *DISTINCT* keyword is equivalent to non-aggregated projection.

4.4 Limit

If the query is an SPJA query, there is no need to extract l_E since there can be only *one* row in any populated result. But in the general SPJGAOL case, the only way to extract l_E is to generate a database instance such that $\mathcal{F}$ produces more than l_E rows in the result R, subject to a maximum limit imposed by the GROUP BY clause.

The number of different values a column can legitimately take is a function of multiple parameters – data type, filter predicates, database engine, hardware platform, etc. Let $n_1, n_2, n_3, ..$ be the number of different values, after applying domain and filter restrictions, that the functionally-independent attributes $A_1, A_2, A_3, ..$ in G_E can respectively take. This means that there can be a maximum of $n_1 * n_2 * n_3 * ... = l_E^{max}$ groups in the result. Thus, l_E values up to l_E^{max} can be extracted with this approach.

To extract l_E, UNMASQUE iteratively generates database instances such that the result-cardinality follows a geometric progression starting with a rows and having common ratio $r(> 1)$. We set $a = max$ (4,cardinality of R) to be consistent with our extraction requirement for G_E which required a permissible result cardinality of upto 3 rows. And r can be set to a convenient value that provides a good tradeoff between the number of iterations (which will be high with small r) and the setup cost of each iteration (which will be high with large r). In our experiments, $r = 10$ was used. This appears reasonable given that the l_E value is typically a small number in most applications – for instance, in TPC-H, the maximum is 100, and in general, we do not expect the value to be more than a few hundreds at most.

Generating D_{gen} for desired R cardinality

To get n rows in the result prior to the limit kicking in, we generate a database instance with each table having n rows such that the functionally-independent attributes in G_E have a unique permutation of values in each row. Specifically, all the attributes appearing in JG_E are assigned values $(1, 2, 3, ..., n)$ and the other attributes are assigned any value satisfying their filter predicates (if any). If the result of applying $\mathcal{F}$ on this database contains m rows with $m < n$, then we can conclude that LIMIT is in operation and equal to m.

With the above procedure, we finally obtain for Q3:

l_E **= 10**

Extension to non-integral Key attributes

There are various applications (e.g. Wilos [45]) which use non-integral keys as identifier in the database tables. We assume that the domain of each key attribute contains at least two different values.

To handle non-integral keys, the following changes are required:

In Mutation Pipeline, only the join predicate extraction module require changes. In this module, instead of negating the values in one of the components, we choose two different fixed values (say p and q) from the domain of the key attribute and assign p attributes in one component and q to the attributes in the other component.

For every module in Generation Pipeline, we again take two different fixed values (say p and q) from the domain of the key attribute. Then, all the assignments that use fixed value 1 are replaced with value p and all the the assignments that use fixed value 2 are replaced with value q.

5 Experiments

Having described the functioning of the UNMASQUE tool, we now move on to empirically evaluating its efficacy and its efficiency. Our experiments were conducted on a representative suite of twelve SPJGAOL queries based on different queries of the TPC-H benchmark, with the primary changes being the removal of nested queries and UDFs – the specific queries are listed in the Appendix, and are similar in complexity to the Q3 running example. We hereafter refer to them by their associated TPC-H query identifiers.

Each query was passed through a Cpp program that embedded the query in a separate executable. These executables formed the input to UNMASQUE, which has been implemented in Python, and were invoked on the TPC-H database, assuring a populated result.

UNMASQUE's ability to non-invasively extract the above queries was assessed on a 100 GB version of the TPC-H benchmark, and to profile its scaling capacity, also on a 1 TB environment. All the experiments were hosted on a well-provisioned[4] PostgreSQL 11 database platform.

We have also run UNMASQUE on (i) the TPC-DS benchmark with PostgreSQL and 100 GB database version, and (ii) the TPC-H benchmark with SQL Shield encrypted queries on Microsoft SQL Server [36]. The performance results were of a similar nature.

Correctness We compared the $\mathcal{Q}_E$ output by UNMASQUE on the above $\mathcal{Q}_H$ suite with the original queries. Specifically, we verified, both manually and empirically with the automated Checker component of the pipeline, that the extracted queries were *semantically identical* to their hidden sources.

Efficiency The total end-to-end time taken to extract each of the twelve queries on the 100 GB TPC-H database instance is shown in the bar-chart of Figure 12. In addition, the breakup of the primary pipeline contributors to the total time is also shown in the figure.

We first observe that the extraction times are practical for offline analysis environments, with all extractions being completed within an hour. Secondly, there is a wide variation in the extraction times, ranging from a few minutes (e.g. Q_2) to almost an hour (e.g. Q_5). The reason is the presence or absence of the lineitem table in the query – this table is enormous in size (around 0.6 billion rows), occupying about 75% of the database footprint, and therefore inherently incurring heavy processing costs.

Drilling down into the performance profile, we find that the first two modules of the UNMASQUE pipeline, FROM clause (purple color) and MINIMIZER (blue color), take up the lion's share of the extraction time, the remaining modules (red color) collectively completing within a few seconds. For instance, for $Q5$ which consumed around 51.15 minutes overall, FROM and MINIMIZER expended around

[4]Intel Xeon 2.3 GHz CPU, 128GB RAM, 3TB Disk, Ubuntu Linux

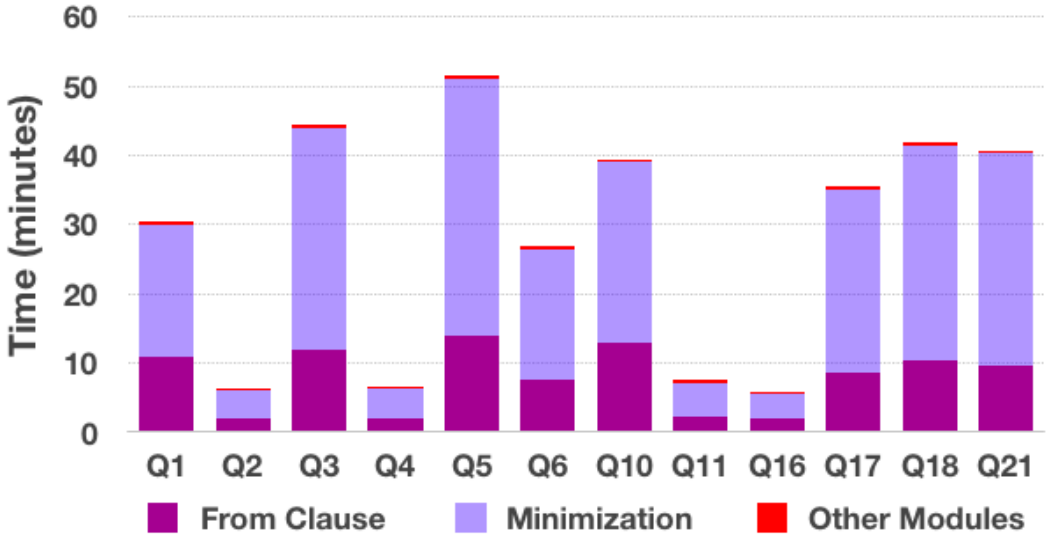

Figure 12: Hidden Query Extraction Time (TPC-H 100 GB)

13 minutes and 38 minutes, respectively, and only a paltry 9 seconds was taken by all other modules combined.

The extreme skew is because these two modules operate on the original large database, whereas, as described in Sections 3 and 4, the remaining modules work on miniscule mutations or synthetic constructions that contain just a handful of rows. Interestingly, although the executable $\mathcal{F}$ was invoked a *few hundred* times during the operation of these modules, the execution times in these invocations was negligible due to the tiny database sizes.

Optimizations We now go on to show how even these initial time-consuming pipeline components could be substantially improved with regard to their efficiency.

Firstly, the FROM clause implementation in the above experiments used the "execution-with-zero-result" approach (Section 3.1) to identify the tables. However, if we choose the alternative "execution-with-error" approach, the identification is completed in just a few seconds!

Secondly, instead of executing MINIMIZER on the *entire* original database, *sampling* methods that are natively available in most database systems could be leveraged as a pre-processor to quickly reduce the initial size. Specifically, we iteratively *sample* the large-sized tables, one-by-one in decreasing size order, until a populated result is obtained. The sampling is done using the following SQL construct:

select * **from** table **where** random() $<$ 0.SZ ;

which creates a random sample that is SZ percent relative to the original table size. An interesting optimization problem arises here – if SZ is set too low, the sampling may require several failed iterations before producing a populated result. On the other hand, if SZ is set too large, unnecessary overheads are incurred even if the sampling is successful on the first attempt. At the current time, we have found a heuristic setting of **SZ = 2%** to consistently achieve both fast convergence (within two iterations) and low overheads. In our future work, we intend to theoretically investigate the optimal tuning of the SZ parameter.

The revised total execution times after incorporating the above two optimizations, are shown in Figure 13, along with the module-wise breakups. We see here that all the queries are now successfully identified in *less than 10 minutes*, substantially lower as compared to Figure 12. Further, the FROM clause takes virtually no time, as expected, and is therefore included in the Other Modules category (green color). And in the MINIMIZER, the preprocessing effort spent on sampling (maroon color) takes

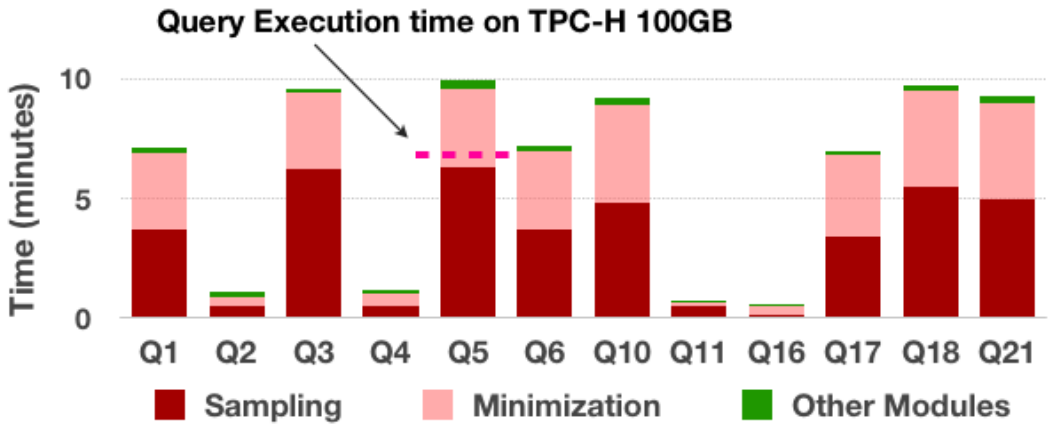

Figure 13: Optimized Hidden Query Extraction Time (TPC-H 100GB)

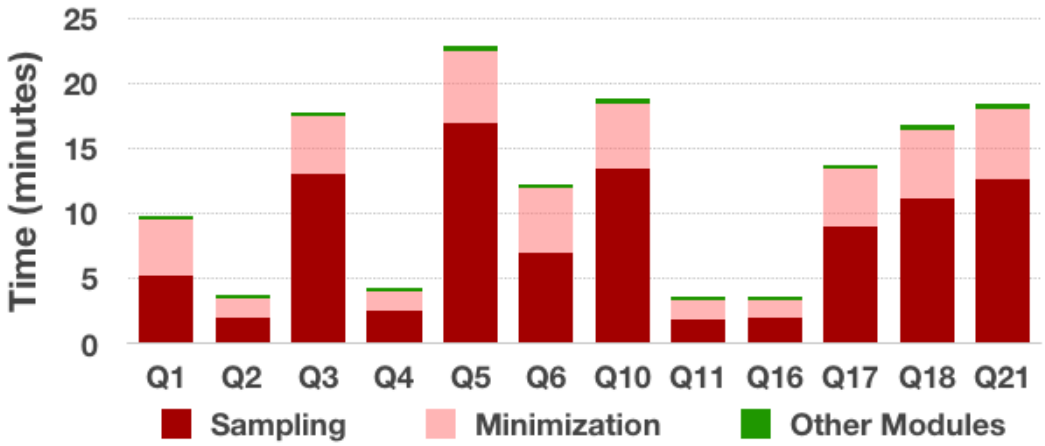

Figure 14: Optimized Hidden Query Extraction Time (TPC-H 1 TB)

the majority of the time, but greatly speeds up the subsequent recursive partitioning (pink color).

An alternative testimonial to UNMASQUE's efficiency is obtained when we compare the total extraction times with their corresponding query response times. For all the queries in our workload, this ratio was less than **1.5**. As a case in point, a single execution of $Q5$ on the 100GB database took around 6.7 minutes, shown by the red dashed line in Figure 13, while the extraction time was just under 10 minutes.

Finally, as an aside, it may be surmised that popular database subsetting tools, such as Jailer [32] or Condenser [43], could be invoked instead of the above sampling-based approach to constructively achieve a populated result. However, this is not really the case due to the following reasons: Firstly, these tools do not scale well to large databases – for instance, Jailer did not even complete on our 100 GB TPC-H database! Secondly, although they guarantee referential integrity, they cannot guarantee that the subset will adhere to the *filter predicates* – due to the hidden nature of the query. So, even with these tools, a trial-and-error approach would have to be implemented to obtain a populated result.

Scaling Profile To explicitly assess the ability of UNMASQUE to scale to larger databases, we also conducted the same set of extraction experiments on a **1 TB** instance of the TPC-H database. The results of these experiments, which included all optimizations, are shown in Figure 14. We see here that all extractions were completed in less than 25 minutes each, demonstrating that the growth of overheads is sub-linear in the database size. In fact, a single query execution of $Q5$ on this database

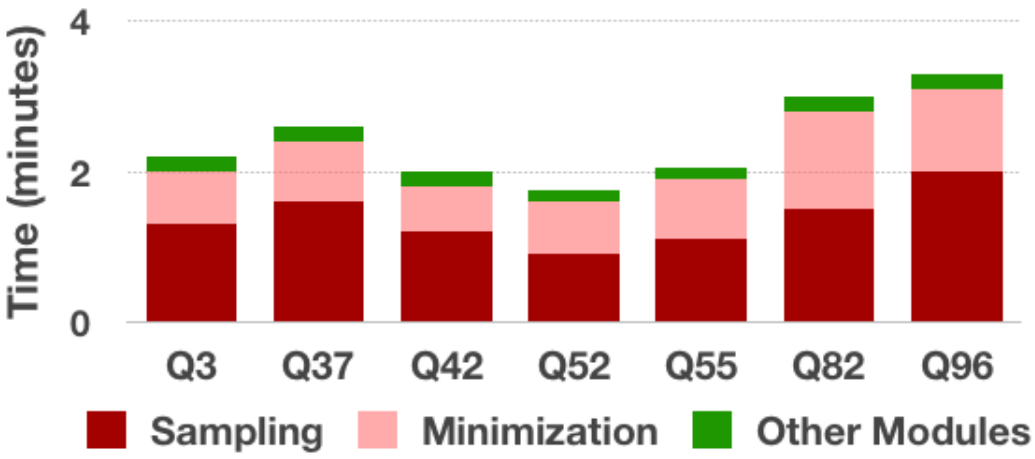

Figure 15: Hidden Query Extraction Time (TPC-DS 100 GB)

took around 72 minutes, almost 3 times the query extraction time. Further, here too an SZ setting of 2 % proved to be a viable choice that provided fast convergence and low overheads.

TPC-DS Results for 100 GB The bar-chart in Figure 15 shows the time taken to extract 7 queries sourced from TPC-DS benchmark (along with there identifier numbers) on a 100 GB database version. The exact queries are listed in Appendix. We can see that all the queries were extracted within 4 minutes. It may surprise at first that the time taken in this case is lesser than the time for TPC-H queries and also, the variation amongst queries is very less. The reason is that the table sizes in TPC-DS are not that skewed as in TPC-H. So, no table in TPC-DS is as huge as lineitem table of TPC-H.

Imperative to SQL Translation We found that for Enki, 14 out of 17 and for blog 2 out of 2 commands were extracted (except insert, update, etc.). Table 3 shows the SQL queries extracted w.r.t. the commands. We have omitted five commands due to lack of space as those were simple table scans. The queries corresponding to remaining three commands did not belong to EQC and only SPJ part was extracted correctly for them. We manually verified that all the commands in table 3 were extracted correctly.

We have included following applications for imperative to SQL translation:

1) **Enki Blogging Application** Enki [33] is an open source application, built with Ruby on Rails, with over 800 stars on GitHub. Enki commands enable the author of the blog to navigate pages, posts, and comments.

2) **Blog**: The Blog application is an example obtained from the Ruby on Rails website [39]. It implements a command that retrieves all articles and a command that retrieves a specific article and its associated comments.

The Enki and Blog servers receive HTTP requests, interact with the database accordingly, and respond the client with an HTML page that contains the data retrieved. Enki uses a total of eight database tables and Blog uses two database tables. We created a synthetic database of 10 MB size which gives non-empty result for each of these commands. Along with UNMASQUE, we used Selenium [35] to send an HTTP request and receive the results in HTML page from which the database results are automatically extracted. We found that for Enki, 14 out of 17 and for blog 2 out of 2 commands were extracted (except insert, update, etc.). Table 3 shows the SQL queries extracted w.r.t. the commands. We have omitted five commands as those were simple table scans. The queries corresponding to remaining three commands did not belong to EQC and only SPJ part was extracted correctly for them. We manually

Command	Application	Extracted SQL Complexity	Time
get admin comments	Enki	Project, Join, OrderBy, Limit	1.2 sec
get admin pages	Enki	Project, OrderBy, Limit	1 sec
get admin pages id	Enki	Select, Project, Limit	1 sec
get admin posts	Enki	Project, Join, GroupBy, OrderBy, Limit	2.5 sec
get admin posts id	Enki	Select, Project, Limit	1 sec
get admin comments id	Enki	Select, Project, Limit	1 sec
get admin undo items	Enki	Project, Order by, Limit	.5 sec
get latest posts	Enki	Select, Project, Join, Filter, GroupBy, Order By, Limit	1.5 sec
get user posts	Enki	Select, Project, Join, Filter, Group By, Order By, Limit	2.5 sec
get latest posts by tag	Enki	Select, Project, Join, Filter, GroupBy, OrderBy, Limit	2.5 sec
get article for id	Blog	Select, Project, join	1 sec

Table 3: Imperative to SQL Translation

verified that all the commands in table 3 were extracted correctly. As a sample instance, consider the *"get latest posts by tag"* command, a snippet of which is outlined in Figure 16a. The corresponding UNMASQUE output is shown in Figure 16b, and was produced in just 2.5 seconds.

```
def find_recent(options = {})
  ...
  include_tags = options[:include] == :tags
  order = 'published_at DESC'
  conditions = ['published_at < ?', Time.zone.now]
  limit = options[:limit] ||= DEFAULT_LIMIT
  result = Post.tagged_with(tag)
  result = result.where(conditions)
  result = result.includes(:tags) if include_tags
    ...
End
```

(a) **Imperative Function Code (snippet)**

```
Select min(posts.title), min(posts.body), max(posts.published_at),
count(*), min(tags.name)
From posts, comments, taggings, tags
Where posts.id = comments.post_id and taggings.tag_id = tags.id
and posts.id = taggings.taggable_id and taggings.taggable_type =
'Post' and (posts.published_at < cur_timestamp)
group by comments.post_id
order by max(posts.published_at) desc limit 15;
```

(b) **Extracted Query (cur_timestamp is a constant)**

Figure 16: Imperative to SQL Translation

6 Conclusions and Future Work

We introduced and investigated the problem of Hidden Query Extraction, which has a variety of real-world use-cases. As the first step toward solving this problem, we presented the UNMASQUE algorithm, which is based on a combination of database mutation and database generation pipelines. An attractive feature of UNMASQUE is that it is completely non-invasive, facilitating its deployment in a platform-independent manner.

UNMASQUE is capable of identifying a large class of hidden SPJGAOL queries, similar to those present in the decision-support benchmarks. For the most part, the extraction pipeline works on miniscule databases designed to contain only a handful of rows. The effects of these optimizations were visible in our experimental results which demonstrated that query extraction could be completed in times comparable with normal query response times in spite of a large number of executable invocations.

A natural question to ask at this point is whether it appears feasible to extend the scope of our extraction process to a broader range of common SQL constructs – for instance, outer-joins, disjunctions and nested queries. As mentioned previously, none of these constructs are handled by the current set of QRE tools. However, based on some preliminary investigation, it appears that outer-joins and disjunctions could eventually be extracted under some restrictions – for instance, the IN operator can be

handled if it is known that the database includes all constants that appear in the clause.

In our current work, we are attempting to extend the scope of EQC to include the Having clause on groups, a common construct in real-world applications. Also, a mathematical analysis to help choose the appropriate SZ setting for sampling. For the long-term, the extraction of nested queries and outer joins poses a formidable challenge. More fundamentally, characterizing the extractive power of non-invasive techniques is an open theoretical problem.

References

[1] A. Bonifati, R. Ciucanu and S. Staworko. Learning Join Queries from User Examples. *ACM TODS*, 40(4), 2016.

[2] B. Chandra. Automated Testing and Grading of SQL Queries. *PhD Thesis, CSE, IIT Bombay*, 2019.

[3] M. Chavan, R. Guravannavar, K. Ramachandra and S. Sudarshan. DBridge: A program Rewrite Tool for Set-Oriented Query Execution. In *Proc. of IEEE ICDE Conf.*, 2011.

[4] A. Cheung, A. Solar-Lezama and S. Madden. Optimizing Database-Backed Applications with Query Synthesis. In *Proc. of ACM PLDI Conf.*, 2013.

[5] P. da Silva. SQUARES : A SQL Synthesizer Using Query Reverse Engineering. *Master's Thesis*, Tecnico Lisboa, Nov 2019. `web.ist.utl.pt/ist181151/81151-pedro-silva_dissertacao.pdf`

[6] R. DeMillo, R. Lipton and F. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4), 1978.

[7] K. Emani, K. Ramachandra, S. Bhattacharya and S. Sudarshan. Extracting Equivalent SQL from Imperative Code in Database Applications. In *Proc. of ACM SIGMOD Conf.*, 2016.

[8] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proc. of ACM POPL Conf.*, 2011.

[9] R. Guravannavar, S. Sudarshan, A. Diwan and Ch. Babu. Which sort orders are interesting?. *The VLDB Journal* 21, 2012.

[10] D. Kalashnikov, L. Lakshmanan and D. Srivastava. FastQRE: Fast Query Reverse Engineering. In *Proc. of ACM SIGMOD Conf.*, 2018.

[11] G. Karvounarakis, Z. Ives and V. Tannen. Querying Data Provenance. In *Proc. of ACM SIGMOD Conf.*, 2010.

[12] W. Kim. On Optimizing an SQL-like Nested Query *ACM TODS*, 7(3), 1982.

[13] H. Li, C. Chan and D. Maier. Query from Examples: An Iterative, Data-Driven Approach to Query Construction. *The VLDB Journal*, 8(13), 2015.

[14] K. Panev and S. Michel. Reverse Engineering Top-k Database Queries with PALEO. In *Proc. of EDBT Conf.*, 2016.

[15] K. Ramachandra, K. Park, K. Emani, A. Halverson, C. Galindo-Legaria and C. Cunningham. Froid: Optimization of Imperative Programs in a Relational Database. *PVLDB*, 11(4), 2017.

[16] S. Shah, S. Sudarshan, S. Kajbaje, S. Patidar, B. Gupta and D. Vira. Generating test data for killing SQL mutants: A constraint-based approach. In *Proc. of IEEE ICDE Conf.*, 2011.

[17] J. Shen and M. Rinard. Using Active Learning to Synthesize Models of Applications that Access Databases. In *Proc. of ACM PLDI Conf.*, 2019.

[18] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding and L. Novik. Discovering Queries based on Example Tuples. In *Proc. of ACM SIGMOD Conf.*, 2014.

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

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

[21] Q. Tran, C. Chan and S. Parthasarathy. Query by Output. *Technical Report, National Univ. of Singapore*, 2009.

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

[23] J. Tuya, M. Cabal and C. Riva. Full predicate coverage for testing SQL database queries. *Software Testing Verification and Reliability*, 20(3), 2010.

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

[25] W. Wong, B. Kao, D. Cheung, R. Li and S. Yiu. Secure query processing with data interoperability in a cloud database environment. In *Proc. of ACM SIGMOD Conf.*, 2014.

[26] M. Zhang, H. Elmeleegy, C. Procopiuc, and D. Srivastava. Reverse Engineering Complex Join Queries. In *Proc. of ACM SIGMOD Conf.*, 2013.

[27] S. Zhang and Y. Sun. Automatically Synthesizing SQL Queries from Input-Output Examples. In *Proc. of IEEE ASE*, 2013.

[28] `www.bleepingcomputer.com/news/security/massive-wave-of-mongodb-ransom-atta`
`makes-26-000-new-victims/`

[29] `www.cybersecurity-insiders.com/ransomware-hits-mysql-servers/`

[30] `www.hibernate.org`

[31] `www.imperva.com/blog/database-attacks-sql-obfuscation/`

[32] Jailer: `www.jailer.sourceforge.net/home.htm`

[33] `https://github.com/xaviershay/enki`

[34] `docs.microsoft.com/en-us/sysinternals/downloads/strings`

[35] `https://www.selenium.dev/`

[36] `www.microsoft.com/en-in/sql-server`

[37] `www.mysql.com/`

[38] `www.postgresql.org`

[39] `https://rubyonrails.org/`

[40] www.red-gate.com/blog/database-devops/database-subsetting-wed-love-hear

[41] www.softwareheritage.org/mission/software-is-fragile

[42] www.sql-shield.com

[43] Condensor: www.tonic.ai/post/condenser-a-database-subsetting-tool/

[44] www.tpc.org

[45] Wilos: an orchestration process software. www.openhub.net/p/6390

Appendix

Experiment Queries 1 (Based on corresponding TPC-H queries)

Q_1

Select l_returnflag, l_linestatus, sum(l_quantity) **as** sum_qty, sum(l_extendedprice) **as** sum_base_price, sum(l_discount) **as** sum_disc_price, sum(l_tax) **as** sum_charge, avg(l_quantity) **as** avg_qty, avg(l_extendedprice) **as** avg_price, avg(l_discount) **as** avg_disc, count(*) **as** count_order
From lineitem
Where l_shipdate $\leq$ date '1998-12-01' - interval '71 days'
Group By l_returnflag, l_linestatus
Order by l_returnflag, l_linestatus;

Q_2

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

Q_3

Select l_orderkey, sum(l_extendedprice) **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 $<$ date '1995-03-15' **and** l_shipdate $>$ date '1995-03-15'
Group By l_orderkey, o_orderdate, o_shippriority
Order by revenue **desc**, o_orderdate
Limit 10;

Q_4

Selecto_orderdate, o_orderpriority, count(*) **as** order_count
From orders
Where o_orderdate $\geq$ date '1997-07-01' **and** o_orderdate $<$ date '1997-07-01' + interval '3' month
Group By l_orderkey, o_orderdate, o_orderpriority
Order by o_orderpriority
Limit 10;

Q_5

Select n_name, sum(l_extendedprice) **as** revenue
From customer, orders, lineitem, supplier, nation, region
Where c_custkey = o_custkey **and** l_orderkey = o_orderkey **and** l_suppkey = s_suppkey **and** c_nationkey = s_nationkey **and** s_nationkey = n_nationkey **and** n_regionkey = r_regionkey **and** r_name = 'MIDDLE EAST' **and** o_orderdate $\geq$ date '1994-01-01' **and** o_orderdate $<$ date '1994-01-01' + interval '1' year

Group By n_name
Order by revenue **desc**
Limit 100;

Q_6

Select l_shipmode, sum(l_extendedprice) **as** revenue
From lineitem
Where l_shipdate $\geq$ date '1994-01-01' **and** l_shipdate $<$ date '1994-01-01' + interval '1' year **and** l_quantity $<$ 24
Group By l_shipmode
Limit 100;

Q_{10}

Select c_name,, sum(l_extendedprice) **as** revenue, c_acctbal, n_name, c_address, c_phone, c_comment
From customer, orders, lineitem, nation
Where c_custkey = o_custkey **and** l_orderkey = o_orderkey **and** o_orderdate $\geq$ date '1994-01-01' **and** o_orderdate $<$ date '1994-01-01' + interval '3' month **and** l_returnflag = 'R' **and** c_nationkey = n_nationkey
Group By c_name, c_acctbal, c_phone, n_name, c_address, c_comment
Order by revenue **desc**
Limit 20;

Q_{11}

Select ps_COMMENT, sum(ps_availqty) **as** value
From partsupp, supplier, nation
Where ps_suppkey = s_suppkey **and** s_nationkey = n_nationkey **and** n_name = 'ARGENTINA'
Group By ps_COMMENT
Order by value **desc**
Limit 100;

Q_{16}

Select p_brand, p_type, p_size, count(ps_suppkey) **as** supplier_cnt
From partsupp, part
Where p_partkey = ps_partkey **and** p_brand = 'Brand#45' **and** p_type **Like** 'SMALL PLATED%' **and** p_size $\geq$ 4
Group By p_brand, p_type, p_size
Order by supplier_cnt **desc**, p_brand, p_type, p_size;

Q_{17}
Select AVG(l_extendedprice) as avgTOTAL
From lineitem, part
Where p_partkey = l_partkey **and** p_brand = 'Brand#52' **and** p_container = 'LG CAN';

Q_{18}

Select c_name, o_orderdate, o_totalprice, sum(l_quantity)
From customer, orders, lineitem
Where c_phone **Like** '27-_%' **and** c_custkey = o_custkey **and** o_orderkey = l_orderkey
Group By c_name, o_orderdate, o_totalprice
Order by o_orderdate, o_totalprice **desc**
Limit 100;

Q_{21}

Select s_name, count(*) **as** numwait
From supplier, lineitem l1, orders, nation
Where s_suppkey = l1.l_suppkey **and** o_orderkey = l1.l_orderkey **and** o_orderstatus = 'F' **and** s_nationkey = n_nationkey **and** n_name = 'GERMANY'
Group By s_name
Order by numwait **desc**, s_name
Limit 100;

Experiment Queries 2 (Based on corresponding TPC-DS queries)

Q_3

Select dt.d_year ,item.i_brand_id as brand_id ,item.i_brand as brand ,sum(ss_sales_price) as sum_agg
From date_dim dt ,store_sales ,item
Where dt.d_date_sk = store_sales.ss_sold_date_sk **and** store_sales.ss_item_sk = item.i_item_sk **and** item.i_manufact_id = 816 **and** dt.d_moy=11
Group By dt.d_year ,item.i_brand ,item.i_brand_id
Order by dt.d_year ,sum_agg **desc** ,brand_id
Limit 100 ;

Q_{37}

Select i_item_id ,i_item_desc ,i_current_price
From item, inventory, date_dim, catalog_sales
Where i_current_price **between** 45 **and** 45 + 30 **and** inv_item_sk = i_item_sk **and** d_date_sk=inv_date_sk **and** d_date **between** date '1999-02-21' **and** date '1999-04-23' **and** i_manufact_id **between** 707 **and** 1000 **and** inv_quantity_on_hand **between** 100 **and** 500 **and** cs_item_sk = i_item_sk
Group By i_item_id,i_item_desc,i_current_price
Order by i_item_id
Limit 100 ;

Q_{42}

Select dt.d_year ,item.i_category_id ,item.i_category ,sum(ss_ext_sales_price)
From date_dim dt ,store_sales ,item
Where dt.d_date_sk = store_sales.ss_sold_date_sk **and** store_sales.ss_item_sk = item.i_item_sk **and** item.i_manager_id = 1 **and** dt.d_moy=11 **and** dt.d_year=2002
Group By dt.d_year ,item.i_category_id ,item.i_category
Order by sum(ss_ext_sales_price) **desc**,dt.d_year ,item.i_category_id ,item.i_category

Limit 100 ;

Q_{52}

Select dt.d_year ,item.i_brand_id as brand_id ,item.i_brand as brand ,sum(ss_ext_sales_price) as ext_price
From date_dim dt ,store_sales ,item
Where dt.d_date_sk = store_sales.ss_sold_date_sk **and** store_sales.ss_item_sk = item.i_item_sk **and** item.i_manager_id = 1 **and** dt.d_moy=12 **and** dt.d_year=2002
Group By dt.d_year ,item.i_brand ,item.i_brand_id
Order by dt.d_year ,ext_price **desc** ,brand_id
Limit 100 ;

Q_{55}

Select item.i_brand_id as brand_id ,item.i_brand as brand
,sum(ss_ext_sales_price) as ext_price
From date_dim dt ,store_sales ,item
Where dt.d_date_sk = store_sales.ss_sold_date_sk **and** store_sales.ss_item_sk = item.i_item_sk **and** item.i_manager_id = 1 **and** dt.d_moy=12 **and** dt.d_year=2002
Group By dt.d_year ,item.i_brand ,item.i_brand_id
Order by ,ext_price **desc** ,brand_id
Limit 100 ;

Q_{82}

Select i_item_id ,i_item_desc ,i_current_price
From item, inventory, date_dim, store_sales
Where i_current_price **between** 45 **and** 45 + 30 **and** inv_item_sk = i_item_sk **and** d_date_sk=inv_date_sk **and** d_date **between** date '1999-07-09' **and** date '1999-09-09' **and** i_manufact_id **between** 169 **and** 639 **and** inv_quantity_on_hand **between** 100 **and** 500 **and** ss_item_sk = i_item_sk
Group By i_item_id,i_item_desc,i_current_price
Order by i_item_id
Limit 100 ;

Q_{96}

Select count(*)
From store_sales ,household_demographics ,time_dim, store
Where ss_sold_time_sk = time_dim.t_time_sk **and** ss_hdemo_sk = household_demographics.hd_demo_sk **and** ss_store_sk = s_store_sk **and** time_dim.t_hour = 8 **and** time_dim.t_minute $\geq$ 30 **and** household_demographics.hd_dep_count = 3 **and** store.s_store_name = 'ese'
Order by count(*)
Limit 100;