Reverse Engineering Aggregation Queries

Wei Chit Tan[1] Meihui Zhang[1*] Hazem Elmeleegy[2] Divesh Srivastava[3]
[1]Singapore University of Technology and Design, [2]Turn Inc., [3]AT&T Labs-Research
weichit_tan@mymail.sutd.edu.sg, meihui_zhang@sutd.edu.sg,
hazem.elmeleegy@turn.com, divesh@research.att.com

ABSTRACT

Query reverse engineering seeks to re-generate the SQL query that produced a given query output table from a given database. In this paper, we solve this problem for OLAP queries with group-by and aggregation. We develop a novel three-phase algorithm named REGAL [1] for this problem. First, based on a lattice graph structure, we identify a set of group-by candidates for the desired query. Second, we apply a set of aggregation constraints that are derived from the properties of aggregate operators at both the table-level and the group-level to discover candidate combinations of group-by columns and aggregations that are consistent with the given query output table. Finally, we find a multi-dimensional filter, i.e., a conjunction of selection predicates over the base table attributes, that is needed to generate the exact query output table. We conduct an extensive experimental study over the TPC-H dataset to demonstrate the effectiveness and efficiency of our proposal.

1. INTRODUCTION

The ability to reverse engineer SQL queries has been established as one way of improving database usability and simplifying the interaction between humans and databases. The most common use case is when a user, Alice, has access to some historical reports (or query results) that were generated by an employee who may have left the organization, or by an application that is no longer in-use or maintained, and hence the actual queries used to generate those results are not available anymore. If Alice needs to re-run those queries to get a fresher version of the reports, or just get a better understanding of why the values in the reports look the way they do, then she needs a tool that can reverse engineer the results to infer the most plausible queries generating them from a given database instance.

This is a real-world scenario that has been referred to in several previous works (e.g., [4, 14, 15, 16]). Other use cases include getting alternative characterizations for a query result, answering why-not questions (or identifying the changes that need to be applied to a query for its result to include a given missing tuple), and even database security applications [14, 15].

Given the prevalence of OLAP-style aggregation queries in data analytics and visualization applications running on databases and data warehouses, consisting of a wide variety of possible query expressions (with multiple group by columns, aggregates, and selection conditions), it is useful to have general and efficient techniques to reverse engineer this important class of queries. This need constitutes the motivation for our work.

Previous work on query reverse engineering has mainly focused on SPJ queries [4, 14, 15, 16], and did not give as much attention to OLAP-style aggregation queries. The few works [8, 15] that considered aggregation queries offered only basic solutions to limited versions of the problem. PALEO [8] focused on top-k queries with an aggregate used as a ranking criterion. The QRE system [15] considered union (SPJU) and aggregation (SPJA) queries. However, both works are restricted in scope to query output tables with a single group-by column and a single aggregate column. Even though QRE [15] had initiated reverse engineering of queries that contain group-bys and aggregations, there are many limitations in their work, as we discuss next.

First, their search space for discovering queries is determined by the schema complexity, the number of selection predicates and the set of attributes used to generate the selection predicates. To reduce complexity, they use control knobs for each possible determinant. For example, the number of relations is controlled such that the schema subgraphs would not be too large and complex.

Second, they do not propose any efficient pruning strategy for aggregation candidates. A column in the query output table may relate to many different aggregate functions especially when there are some selection predicates applied. For instance, a `MAX` over a small subset of tuples is also likely to be equal to the `MIN`, `SUM`, or `AVG` over another subset of tuples in certain groups. The problem is compounded when there are aggregate functions associated with multiple output columns. Hence, the candidate aggregation query generation could be more effective if an aggregation constraint pruning strategy is used effectively.

Third, their data classification method TALOS [14, 15] uses a decision tree to find the suitable selection predicates and does not guarantee that the complexity of returned queries is minimum. Intuitively, the number of leaf nodes in a decision tree reflects the number of selection predicates and the node splits are computed based on the class label assignments for the tuples in the joined relations. In addition, the node splits depend on the set of attributes used for the decision tree construction, and hence the aggregation queries that are generated might have more complex selection conditions than necessary.

*Contact author.

[1]REGAL: Reverse Engineering Group-bys, Aggregates and seLections

Proceedings of the VLDB Endowment, Vol. 10, No. 11

Finally, due to the above limitations, the query output table used by [15] only contains two columns, of which one represents the group-by column while the other is an aggregation column. In contrast, we want to solve the general problem by allowing a query output table with many columns, some of which are grouping attributes while the others are aggregations.

1.1 Contributions

In this paper, we propose an efficient algorithm to reverse engineer group-by and aggregation queries. To focus on the novel challenges that group-by and aggregation imposes on query reverse engineering, we do not consider SQL queries with joins, and instead we assume we are given a single base table $\mathcal{B}$ and a query output table Out which is generated from $\mathcal{B}$ by an SQL query with group-by, aggregation and optionally certain selection predicates as filters. The goal is to reconstruct a query that can produce the same table Out from $\mathcal{B}$. We return the queries with the smallest complexity in terms of the dimensionality of the filters, among all group-by and aggregation queries that can produce the table Out. We make the following key contributions to achieve this goal.

- First, we propose a lattice exploration strategy that can discover group-by candidates by efficiently pruning the invalid candidates in the lattice.
- Second, we present a set of effective rules at both the table- and group-level for quickly invalidating incorrect aggregation candidates. We also identify relationships between aggregation functions that could help quickly prune invalid combinations of aggregations.
- Third, we design an efficient algorithm to find the selection conditions (if any) by searching for fuzzy bounding boxes in multi-dimensional matrices, whose dimensions are drawn from the base attributes. Each matrix corresponds to a specific group and a specific aggregation function where all tuples found in the fuzzy bounding box (to be generated) will generate the aggregation result that corresponds to the group. As there could be multiple groups and aggregation functions in one query, we develop a cross validation strategy to prune the invalid fuzzy bounding boxes and refine the remaining candidates until the filter is found.
- Finally, we run experiments on the TPC-H benchmark. We compare our algorithm with the existing QRE solution. Besides, we carefully study each phase of our algorithm by investigating different alternatives. We empirically show that our algorithm is efficient and effective in practice.

1.2 Related Work

The notion of reverse engineering for SQL queries is inspired by QBO [14], VDP [5] and QRE [15]. The Query by Output (QBO) approach [14] discovers a set of instance-equivalent queries (IEQs) for SPJ (select-project-join) queries where the IEQ's output will be equivalent to the given input query at the tuple-instance level. On the other hand, the view definition problem (VDP) [5] focuses on the succinctness of the discovered queries. These discovered queries can have more general selection conditions in order to augment the quality of query reconstruction. The Query Reverse Engineering (QRE) approach [15] considers more alternatives in the initial problem setting of QBO [14]. First, the input query is set to be unknown so that finding IEQs is only dependent on the query output table and the source database. Second, instead of simple SPJ queries, QRE discovers IEQs for more classes of SQL queries which include unions, group-bys and aggregations. Moreover, the source database can be varied as it will be updated from time to time and stored in different versions. Therefore, it is essential for the users to know the exact database version to discover all possible IEQs. Zhang et al. [16] propose a solution to reverse engineer complex join queries with arbitrary join graphs, but without group-by and aggregations.

PALEO [8] discovers top-k SQL queries based on the given ranked query output table. Each discovered top-k query should contain a group-by clause with an order-by clause which is a single aggregation. The follow-up work [9] to PALEO improves their original approach by only requiring the users to input the top-k entities without the corresponding values. These works consider group-by, aggregation and selection conditions. However, both works have serious practical limitations such as being limited to query output tables with a single group-by column and a single aggregate column. Also, PALEO relies on external sources (wikipedia tables) and probabilistic models to reduce the search space, making the assumption that the database vocabulary strongly overlaps with the tables in wikipedia. Further, the attributes used for selection conditions must be categorical and without duplicate values, and the candidate predicates are a conjunction of equality predicates. In contrast, without such strong assumptions, our work provides a more general and more efficient solution to discovering the aggregation queries in a much larger search space.

The problem of query discovery by using examples in an RDBMS has been considered extensively in the literature. First, Query by Example (QBE) [17] converts a user's example elements into SQL format. QBE provides ease of querying for those users who are not familiar with SQL queries. However, they must know the database schema in order to pose a proper example to construct the queries. To relax this constraint of having to have an understanding of the schema, several works on keyword search over relational databases have been developed such as DISCOVER [6], DBXplorer [3] and many others. The basic idea of these systems is to provide a set of keywords; tuples that contain at least one keyword are interconnected to form a joining network that depicts the foreign/primary key relationships among them which is considered a potential answer. Further, keyword search is extended into user-input sample instances, so that schema mappings, in the form of project-join queries over the source database, can be discovered through the sample-driven schema mapping system by Qian et al. [11]. This approach is later extended by Shen et al. [13], which takes an example table that consists of a few tuples (either completely or partially filled) in lieu of using a sample tuple, and the system will return the minimal project-join queries as output.

Since query reverse engineering approaches such as QBO [14] return a set of candidate queries, Query from Examples (QFE) [7] offers an interactive approach to look for the desired query wherein a user can interactively determine her target query by checking modified database-result pairs; this feedback from the user is essential for the system to minimize the number of iterations.

Learning join queries from user examples [4] demonstrates the framework of inference of joins through user interactions by labelling tuples without accessing the integrity constraints. Instead of a complete query output table from an original unknown query, users only need to label some tuples as they may want them to be either included or excluded in the query output table. Since the work is based on user interaction, it is imperative to develop a strategy that proposes the informative tuples for users to minimize their efforts to label tuples in order to find their intended join queries.

Related to query reverse engineering, relational learning [10] is a research direction that uses machine learning techniques to learn different schemas under the same data set. Relational learning aims to yield schema independence for databases to return the same outputs under different schemas. In relational learning, given input

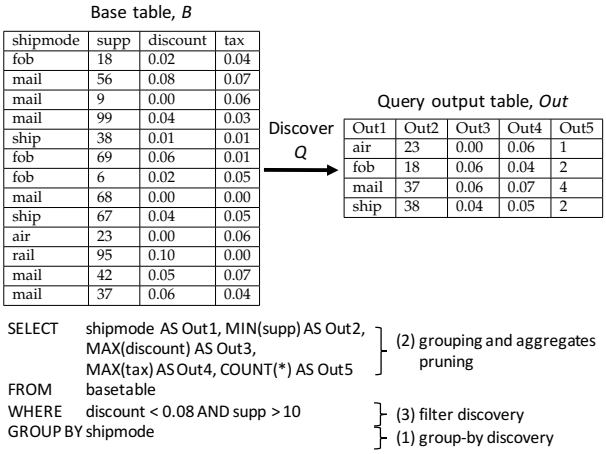


Figure 1: An illustrative example with base table, query output table and SQL query.

training data which contains positive and negative examples, background knowledge and a target relation, the algorithm will learn a hypothesis that is consistent with the background knowledge, to generate the target relation.

1.3 Paper Organization

The remainder of the paper is organized as follows. Section 2 provides a motivating example. Section 3 discusses the preliminaries of our algorithm. Section 4 describes the details and the optimizations of each phase in our algorithm. Experimental evaluation is discussed in Section 5. We conclude our work in Section 6.

2. MOTIVATING EXAMPLE

Consider the base table and query output table in Figure 1. It is easy to see the query output table cannot be generated by a simple select-project query since the values in $Out5$ are not present in any base column, but are more like a count aggregate. The search space must then be extended to queries with group-by and aggregations. Further, one query attribute can be mapped to many possible base table attributes and many query attributes can be mapped to the same possible base table attribute. For instance, $Out3$ can be from either tax or $discount$, and both $Out3$ and $Out4$ can be mapped to tax. This makes the problem of resolving the relationship between the base table and query output table very challenging.

In order to identify the SQL query where we expect that both group-by and aggregations exist, the first problem is to identify what groups are present in the query output table. We notice that the cardinality of the query output table indicates the number of groups, and the group-by attributes (as the group identifier) in the query output table should be unique and contained in the base table. Therefore, with the prerequisites above, other than $Out5$, all the other projected columns and their unique combinations have the possibility to be the group identifier of the query output table. For example, to examine whether the set of projected attributes {*Out1, Out2, Out3, Out4*} can be considered as a group identifier, we first notice that all its tuples in query output table Out are unique but it could not be the group identifier because the tuple (`fob`, `18`, `0.06`, `0.04`) is not contained in the base table. Based on this observation, we perform keyness and containment checkings to identify the valid group-by candidates. Specifically, to reduce the checking times, we construct a lattice graph where the root node consists of the maximum number of possible group-by attributes, and an edge from parent to child node indicates the superset-subset relationship. Once the lattice graph is constructed, the group-by key candidates for the query can be discovered through keyness and containment checkings. Novel lattice traversal algorithms are designed to speed up the discovery process.

The next critical step is to determine the possible aggregations that are used to produce the query output table w.r.t. the group-by attributes generated from the previous step. However, determining a potential valid aggregation for a projected column is not easy because there are too many possibilities. The complex mapping structure is one of the contributing factors for the problem. For instance, we could not resolve whether the mapped base attribute for $Out3$ is tax or $discount$. Therefore, it is likely we will have more false positive aggregations for $Out3$ compared to those query attributes with unique mappings such as $Out2$ which is only mapped to $supp$. Hence, we exploit the properties of aggregation constraints in order to quickly validate a candidate aggregation. In addition, these aggregation constraints are not only applied for single columns, but also for combinations of columns. Given a pair of columns, e.g. $Out3$ and $Out4$, we can scan through their pairwise tuples to uncover whether there is a constant relationship between them, e.g. if the values in $Out3$ are always larger than those in $Out4$.

Another factor that impacts the possibilities of candidate aggregations is the existence of selection conditions. Therefore, we need to determine a subset of tuples that can produce all the aggregate values in the query output table. For instance, only four out of six base tuples, which contain the value $mail$ are used to compute the value of $Out5$ corresponding to $mail$ which is equal to 4. The same applies for other aggregate values in the query output table. We use a matrix structure to solve the problem. Starting from a single base attribute as dimension, we build matrices for every group and aggregate in the query output table. Two search algorithms are designed to find feasible regions (represented as fuzzy bounding boxes) in the matrices. The generated regions are then validated across different groups and aggregation candidates. The algorithm will search for possible regions in a higher dimension if no feasible solutions are found in lower dimensions. In this way, we return the query with the smallest complexity of the selection conditions.

3. PRELIMINARIES

In this paper we assume we are given a single base table $\mathcal{B}$ and a query output table Out which is generated from $\mathcal{B}$ by an aggregation query. The focus is to discover the group-by and aggregation components with the possibility that certain selection conditions are applied. For the queries that involve joins and the projected columns are from multiple tables, we can apply the prior work from Zhang et al. [16] on these projected columns in Out during a preprocessing phase to discover the join path so that a single base table could be generated as the input to this work. Optimizations could also be investigated which we leave for future work.

Aggregation query. Let $\mathcal{Q}$ be the aggregation query that will produce query output table Out from base table $\mathcal{B}$. Let $\mathcal{A}$ denote the set of base attributes where $\mathcal{A} = schema(\mathcal{B})$. The aggregation query $\mathcal{Q}$ can be expressed in the SQL form:

```
SELECT    A_1 AS Out_1 , ... , A_k AS Out_k,
          F_1 AS Out_{k+1} , ... , F_m AS Out_{k+m}
FROM      B
WHERE     C_1 ∧ ... ∧ C_ℓ
GROUP BY  A_1, ..., A_k
```

where $Out_i, i \in [1,k]$ are the group-by attributes in the query output table generated from $A_i, i \in [1,k]$, and $Out_i, i \in [k+1, k+m]$ are the aggregation attributes that are computed by $\mathcal{F}_i, i \in [1,m]$. Each aggregation function $\mathcal{F}_i = AGG(A)$ is built from a single

Algorithm 1: REGAL Algorithm

input : $\mathcal{B}$: base table, Out: query output table, ϕ: mapping table
output: $\mathcal{Q}$: aggregation query

//Phase 1: Group-by discovery
$\mathcal{G}(\mathcal{V}, \mathcal{E}) \leftarrow$ LatticeConstruction(ϕ)
foreach *mapping* ϕ **do**
  set of nodes $\{\mathcal{V}\} = \emptyset$
  $\{\mathcal{V}\} \leftarrow$ KeynessChecking($\mathcal{G}$)
  $\{\mathcal{V}\} \leftarrow$ ContainmentChecking($\mathcal{G}$)
update (ϕ, $\{\mathcal{V}\}$)
//Phase 2: Grouping and aggregates pruning
foreach *column* $\lambda \in Out$ **do**
  set of aggregations $\{\mathcal{F}\} = \emptyset$
  $\{\mathcal{F}\} \leftarrow$ TableConstraints(λ, ϕ)
update (λ, $\{\mathcal{F}\}$)
foreach *mapping* ϕ **do**
  map$< \mathcal{V}, F >= \emptyset$
  foreach *node* $\mathcal{V}$ **do**
    set of aggregations $\{\mathcal{F}\} \leftarrow$ GroupConstraints($\mathcal{V}$, λ, $\{\mathcal{F}\}$)
    $< \mathcal{V}, F > \leftarrow$ Multi-ColumnConstraints(λ, $\{\mathcal{F}\}$)
update(ϕ, $< \mathcal{V}, F >$)
//Phase 3: Filter discovery
initialize set of predicates $\{\mathcal{C}\} = \emptyset$ and
dimensionality $\mathcal{N} = 1$
while $\{\mathcal{C}\} \neq \emptyset$ **do**
  foreach *dimensions of* $A_1, ..., A_{\mathcal{N}} \in \mathcal{A}$ **do**
    set of matrices $\mathcal{M} \leftarrow$ Matrix(ϕ, $< \mathcal{V}, F >$)
    set of bounding boxes $\{\Omega\} \leftarrow$ FilterGeneration($\mathcal{M}_1$)
    while $i < |\mathcal{M}|$ **do**
      update $\{\Omega\} \leftarrow$ CrossValidation($\mathcal{M}_i$)
    $\{\mathcal{C}\} \leftarrow \{\Omega\}$
  $\mathcal{N}$++
Queries $\mathcal{Q} \leftarrow$ update(ϕ, $< \mathcal{V}, F >$, $\{\mathcal{C}\}$)

aggregate operator AGG, i.e. `COUNT`, `SUM`, `AVG`, `MAX`, and `MIN`, associated with any attribute $A \in \mathcal{A}$. The selection condition of the query $\mathcal{Q}$ is a conjunction of predicates $\mathcal{C}_1 \wedge ... \wedge \mathcal{C}_\ell$, where each $\mathcal{C}_j, j \in [1, \ell]$ is expressed as A *op* X, a predicate involving a base attribute A, a comparison operator *op*, and a constant value X. In the experiment, we only select the numerical attributes with some basic comparison operators like "$\leq$", "$<$", "$>$" and "$\geq$" for selection predicates. However, our work can be naturally extended to categorical attributes and other comparison operators.

Mapping Table. As a preprocessing step, we construct a column-mapping table ϕ, which indicates for each output column, $Out_i, i \in [1, k+m]$ whether its distinct values can be mapped to the distinct values of any base attribute(s), denoted as $\phi(Out_i) \subseteq \mathcal{A}$. An output column can be mapped to zero, one, or more attributes in the base table and vice versa. Table 1 describes the many-to-many mapping relationships between the base table and the query output table in Figure 1. The output column $Out5$ is not mapped to any attributes in $\mathcal{A}$, hence $\phi(Out5)$ is a null value.

4. QUERY CONSTRUCTION

This section describes our three-phase algorithm that discovers aggregation queries. The overall algorithm is shown in Algorithm 1. The first phase generates the set of potential grouping attributes that characterizes the `GROUP BY` clause. The second phase yields combinations of grouping attributes with aggregations that potentially match the `SELECT` clause in the query. The third phase returns a set of selection predicates that form the `WHERE` clause in the query.

4.1 Phase 1: Group-by Discovery

In order to discover the set of candidates that can be used as grouping attributes in the query, we build a lattice graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ that consists of all possible group-by candidates, and then develop discovery algorithms based on the properties of group-by keys.

Table 1: Mapping table

Out_i	Out1	Out2	Out3	Out4	Out5
$\phi(Out_i)$	shipmode	supp	tax,discount	tax	null

4.1.1 Lattice Construction

Lattice. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote the lattice graph, where each node $V \in \mathcal{V}$ is a group-by candidate and each edge $E \in \mathcal{E}$ from parent to child node indicates the superset- subset relationship. Figure 2 illustrates the lattice graph built for the example in Figure 1. To construct the lattice graph $\mathcal{G}$, we first identify the maximum set of query columns that can be potential group-by attributes. This maximum set will be used as the lattice root. The rest of lattice nodes can then be created subsequently, according to the superset-subset relationship. We notice that the basic requirement for a column to be part of a group-by key is that the values in that column must be contained in the base column. Given the mapping table ϕ, we could easily identify which query columns are contained in the base table. For instance, based on the mapping table in Table 1, all columns except $Out5$ have a mapping column in the base table. Thus, the maximum set of possible group-by attributes is $\{Out1, Out2, Out3, Out4\}$. A lattice rooted at $\{Out1, Out2, Out3, Out4\}$ will be built, and the rest of nodes can be created accordingly. Then, $keyness$ and $containment$ checks are to be done over the lattice to validate the group-by candidates, as discussed below.

4.1.2 Keyness Checking

For a given node V in the lattice, $keyness$ is defined as the requirement that the values of the attributes in V in the query output table are all unique. A valid group-by candidate must pass the keyness checking. In other words, a column (or a combination of columns) containing duplicate values cannot be a group-by key. Considering the example in Figure 1, $Out1$ is a valid group-by candidate since it contains four unique values, whereas $Out3$ is invalid as two `0.06` values are present. To check the keyness for every lattice node, two traversal methods can be deployed.

Top-down keyness checking. Starting the keyness checking from the lattice root, if there is a parent node that did not pass the keyness checking, its children nodes can also be pruned safely. For instance, if the parent node has the tuple set $\{(a, b), (b, c), (a, b)\}$, which contains a duplicate (a, b), its children nodes would have tuple sets $\{a, b, a\}$ and $\{b, c, b\}$ respectively. The individual keyness checking for them will be unnecessary since the duplicates in the parent node will yield duplicates in the children nodes for sure.

Bottom-up keyness checking. Starting with the leaf node in the lattice graph, if the child node passes the keyness checking, its parent nodes will also be passed automatically without any additional checking, which can save a lot of checking time. Conversely, a parent node needs to be checked only when all its children nodes fail. For instance, we never check the parent node of $\{a, b, c\}$ and $\{c, c, d\}$ since any attribute combining with unique set $\{a, b, c\}$ is guaranteed to be unique. We check only when both child nodes, e.g., $\{c, d, c\}$ and $\{c, c, d\}$, fail. Their common parent node with the tuple set $\{(c, c), (d, c), (c, d)\}$ needs to be checked in this case.

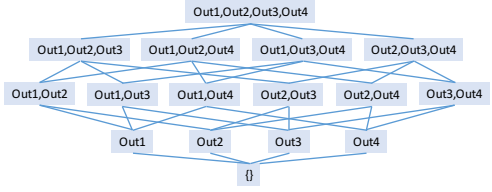


Figure 2: Lattice graph example

4.1.3 Containment Checking

For a given node V in the lattice, *containment* is defined as all the values of the attributes in V in the query output table should be contained in the base table. If V consists of multiple attributes, the combination of the values should be contained. Considering the example in Figure 1, $\{Out1, Out2\}$ with values `(air, 23)`, `(fob, 18)`, `(mail, 37)` and `(ship, 38)` is considered as a valid group-by key candidate since all the value combinations can be found in $\{shipmode, supp\}$ in the base table, whereas $\{Out2, Out3\}$ is invalid as some values, e.g. `(18, 0.06)`, are not contained in the base table, even though each individual column is contained. Similar to keyness checking, there are two possible methods to traverse the lattice for containment checking.

Top-down containment checking. Starting from the lattice root, if a tuple from the corresponding attributes in base table makes a parent node pass the containment checking, its children nodes will also be passed by the same tuple. Therefore, checking time can be saved. Consider our running example in Figure 1. Knowing `(air, 23, 0.00, 0.06)` is contained at node $\{Out1, Out2, Out3, Out4\}$ w.r.t. $\{shipmode, supp, discount, tax\}$, checks at the children nodes can be saved as any subset of values will also be contained in the corresponding base columns. In contrast, if a tuple makes the parent node fail the containment checking, it can be used to check on its children nodes for pruning purpose. We denote such tuples as witness tuples. For instance, the tuple in the query output table `(fob, 18, 0.06)` is not contained in the base table with mapped attributes $\{shipmode, supp, discount\}$. It becomes a witness tuple for checking the containment at the children nodes such as $\{Out1, Out2\}$, $\{Out1, Out3\}$ and $\{Out2, Out3\}$.

Bottom-up containment checking. Here, containment checking starts from the parents of the leaf nodes. This is because the way that we construct the lattice has ensured that all the tuples from leaf nodes are contained in their respective base attributes. If there is a child node whose tuple is not contained, its parent node will be pruned immediately since a superset of a witness tuple will also be a witness. For instance, since the tuple `(0.06, 0.08)` in node $\{Out3, Out4\}$ is not contained in the corresponding base attributes $\{discount, tax\}$ in the base table, this witness tuple would prune the parent nodes such as $\{Out1, Out3, Out4\}$ and $\{Out2, Out3, Out4\}$ automatically.

4.1.4 Optimizations

There are several optimizations for reducing the computation cost of finding all the group-by key candidates.

Bottom-up and top-down strategies. Among all kinds of lattice traversal for keyness and containment checking, we identify that the combination of bottom-up keyness and top-down containment checking is the most effective lattice exploration technique to reduce the number of nodes checked. By starting the keyness checking from leaf nodes, the number of nodes to be checked can be significantly reduced when the leaf nodes have a unique tuple set, in which case the checking for their ancestor nodes can be saved. Further, a node is checked for keyness only when all its children nodes fail the keyness checking. Compared to bottom-up strategy, top-down keyness checking is less effective. First, checking a node at upper level is more expensive then checking a node at lower level since more attributes are involved at upper level. Second, the upper nodes have higher chances to pass the keyness checking. However, we can prune children nodes only when the parent node fails the checking. Therefore, top-down keyness will result in more checking. This is validated in our experiments.

In contrast, containment checking should start from the lattice root. Any witness tuple that is retrieved from the parent node in the query output table and not matched with the respective attributes in the base table can be used to prune its children nodes by doing the containment checking on them. Otherwise, the containment checking on the children nodes can be saved for non-witness tuples. In contrast, bottom-up containment checking requires the ancestor nodes to be checked again even when the child node passes the check. Parent nodes can be pruned only when the child node fails the containment checking. However, the containment checking at lower nodes has higher chances to be successful, which makes the pruning less effective.

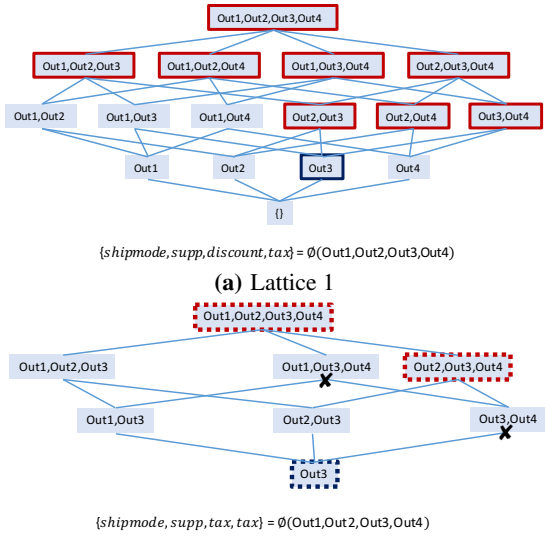


Figure 3: Optimizations in group-by discovery. (a) Group-by candidate nodes in Lattice 1 that are outlined in red are pruned by containment checking while the node outlined in black is pruned by keyness checking. (b) Lattice nodes that are needed for containment checking in Lattice 2 where the dotted nodes are propagated by the failure nodes in Lattice 1 and the crossed out nodes are pruned by overlapped mapping columns.

Propagation of pruned nodes. When the number of mappings between the query output table and base table increases, the number of checks for containment in the lattice graph would also be increased since containment checking requires each tuple of the lattice nodes in the query output table to be checked with its particular mapped base attributes. However, redundant computations may be performed when some similar nodes exist among two mappings. This redundancy can be minimized by propagating the results of the checked nodes into other mappings.

First, for one-to-many and many-to-many mappings, where the query column maps to more than one base attribute, we can avoid the redundant checking on similar nodes that exist in different mappings. Figure 3 shows the lattice root $\{Out1, Out2, Out3, Out4\}$ in the query output table can be mapped to $\{shipmode, supp, discount, tax\}$ and $\{shipmode, supp, tax, tax\}$ respectively. It is a many-to-many mapping as $Out3$ can be mapped to either $discount$ or tax and tax can map to both $Out3$ and $Out4$. Since only $Out3$ is different between the two lattice graphs, therefore any lattice nodes that exhibit the same mapping columns can be propagated from the mapping $\{shipmode, supp, discount, tax\}$ to the mapping $\{shipmode, supp, tax, tax\}$ except the node $Out3$ and all its ancestors. All propagated lattice nodes will be exempted from containment checking, thus are not shown in Figure 3(b).

Second, the impact of the failure nodes can be propagated as well. A node that is deemed as not a group-by key after being invalidated by keyness checking will be exempted from containment checking even if the query column in that node happened to have

a different base attribute mapping. For instance, $Out3$ fails in the keyness checking so it is not needed for containment checking no matter whether its mapping attribute is $discount$ or tax. Besides, the pruning effect of a child node that fails containment checking can also be propagated to its ancestor nodes. For example, the node $\{Out2, Out4\}$ which maps to $\{supp, tax\}$ will cause its ancestor nodes $\{Out2, Out3, Out4\}$ and $\{Out1, Out2, Out3, Out4\}$ to be pruned regardless of the mappings of node $Out3$.

Overlapped mapping columns. There might be cases where a mapping contains multiple instances of the same base column. For example, for mapping $\{shipmode, supp, tax, tax\}$ in Figure 3, the nodes $Out3$ and $Out4$ are mapped to the same base attribute tax. However, according to the properties of a group-by key, the set of output values will be the same if we project the group-by column twice. Therefore, the values in $Out3$ and $Out4$ should be exactly the same, otherwise this is not a valid group-by candidate. And since they are indeed different in our running example, these nodes and their parents are pruned. As shown in Figure 3(b), there are only three remaining lattice nodes left for containment checking out of eight lattice nodes.

4.2 Phase 2: Grouping & Aggregates Pruning

As a consequence of group-by discovery, we judiciously determine candidate sets of base attributes as possible grouping columns in the query output table Out. For each choice of grouping columns obtained from the group-by discovery phase, the next step is to determine the remaining m-combinations of aggregate columns in the query output table Out as $\{\mathcal{F}_1, ..., \mathcal{F}_m\}$. Thus, we can complete the formation of the `SELECT` clause in the to-be-generated SQL query $\mathcal{Q}$. One can imagine a brute-force approach to find these combinations of aggregations. For each group-by candidate, each non-grouping attribute in the query output table has the possibility to be any of the five aggregate operators over any base attribute A. To save cost and minimize the possibilities, we carefully define a series of effective constraint rules based on the properties of each aggregate function.

We divide the constraints we check into single column aggregation constraints and multi-column aggregation constraints. Single column aggregation constraints are further divided into table-level and group-level where group-level constraints will chunk the base table $\mathcal{B}$ into partitions by the already-selected group-by candidate. Similar to column analysis in database profiling [2], the semantic relationship between a query column λ and a base attribute A can be revealed by analyzing their metadata, cardinalities and distinct values. For efficiency, we build an index structure offline (as pre-processing), which captures this type of analysis for each base column to help speed up the checking of aggregation constraints.

4.2.1 Table-level Aggregation Constraints

According to Algorithm 1, for each column λ in query output table Out, each of its column values $\Lambda \in \lambda$ is an aggregate value where a valid aggregation $\mathcal{F}$ should be able to return all these column values.

Before we discuss each unique aggregation constraint, we note that the data type of a column can reveal which constraints are applicable to it. For example, the aggregation $AGG \in \{$`MAX, MIN`$\}$ is compatible with all data types. It requires, however, every aggregate value $\Lambda \in \lambda$ to be present in the base attribute A for it to be a valid candidate. In contrast, the aggregation $AGG \in \{$`SUM, AVG, COUNT`$\}$ is only compatible with numeric data types, and it requires that every aggregate value $\Lambda \in \lambda$ to fall within a certain numeric range for aggregation $AGG(A)$ to be valid. The `COUNT` aggregate can be treated as `SUM` aggregate by computing the summation on a special attribute for each tuple with a value of 1.

Table 2: Table-level aggregation constraints

AGG	Constraint Rules
`COUNT`	If λ is `COUNT(*)`, then (1) $\forall\Lambda \in \lambda$, $\Lambda \in \mathbb{N}$ (2) `SUM`$(\Lambda) \leq$ num_rows$(\mathcal{B})$
`MAX/MIN`	If λ is `MAX`(A)/`MIN`(A) where $A \in \phi(\lambda)$, then $\forall\Lambda \in \lambda$, num_occurrences$(\lambda.\Lambda) \leq$ num_occurrences$(A.\Lambda)$
`AVG`	If λ is `AVG`(A), then $\forall\Lambda \in \lambda$, (1) `MIN`$(A) \leq \Lambda \leq$ `MAX`(A) (2) If $\Lambda =$ `MAX`(A), num_occurrences$(\lambda.\Lambda) \leq$ num_occurrences$(A.\Lambda)$ (3) If $\Lambda =$ `MIN`(A), num_occurrences$(\lambda.\Lambda) \leq$ num_occurrences$(A.\Lambda)$
`SUM`	Given $A = A^+ \cup A^-$ where $A^+ = \{a \in A : a > 0\}$ and $A^- = \{a \in A : a < 0\}$ If λ is `SUM`(A), then $\forall\Lambda \in \lambda$, (1) If $A^+ \neq \emptyset$ and $A^- \neq \emptyset$, `SUM`$(A^-) \leq \Lambda \leq$ `SUM`(A^+) (2) If $A^- = \emptyset$, `MIN`$(A) \leq \Lambda \leq$ `SUM`(A) (3) If $A^+ = \emptyset$, `SUM`$(A) \leq \Lambda \leq$ `MAX`(A)

Given base table $\mathcal{B}$ and query output table where its column $\lambda \in Out$, we discuss each aggregation constraint in Table 2. Table 3 shows all the possible aggregations for each projected column after applying the table-level aggregation constraints (ignore the color labelling for now). For example, $Out5$ can be a count aggregate since its values $\{$`1,2,4,2`$\}$ are all natural numbers and its sum value is less than the number of rows in the base table. Moreover, since $Out3$ is mapped to either $discount$ or tax, and the duplicate value `0.06` also exists in these attributes exactly, therefore the `MAX` and `MIN` aggregates are valid for them. For an example of `AVG` aggregate, $Out2$ whose column values are in the range of `[18,38]`, is matched with base attribute $supp$ whose attribute values are in the interval of `[9,99]`. $Out4$ has the candidate aggregations such as `SUM`(tax) and `SUM`$(discount)$ as it satisfies the constraint rule (2) for `SUM` aggregate that is stated in Table 2.

4.2.2 Group-level Aggregation Constraints

The purpose of the group-level aggregation constraints is to scrutinize the set of possible aggregations $\{\mathcal{F}\}$ by associating the group-by candidates (k-grouping column) in order to finalize a set of combinations of group-by key-aggregation pair $< V, F >$ where $V = \{A_1, ..., A_k\}$ and $F = \{\mathcal{F}_1, ..., \mathcal{F}_m\}$ that can be used in the `SELECT` clause in the query $\mathcal{Q}$. Given each V, the total number of possible $< V, F >$ pairs will be $|V| \times |F|$ where $|F| = |\mathcal{F}_1| \times ... \times |\mathcal{F}_m|$. However, the overall computational cost will be exacerbated as these resulting pairs are all needed for finding suitable predicate(s). Therefore, the group-level aggregation constraints play a crucial role for reducing such intermediate costs.

First, to reduce each $|\mathcal{F}_i|$, $i \in [1, m]$, for each group-by candidate V, we split the base table based on the tuples found in the k-grouping column of the query output table Out. For instance, given $\phi(Out1) = shipmode$ is the group-by candidate, the base table will be grouped into partitions by all $Out1$'s values, namely `air`, `fob`, `mail` and `ship`. The aggregation constraints in Table 2 can be used to prune the invalid aggregations in each set of possible aggregations $\{\mathcal{F}\}$ for each column $\lambda \in Out$ by changing the original base table into each base table partition accordingly.

In addition, if there was an aggregate value Λ that did not satisfy any of the aggregation constraints, and hence could not be explained by any type of aggregation, then we can prune the group-by candidate directly. A heuristic approach to efficiently prune the group-by candidate is to test the aggregate values with the smallest set of possible aggregations $\{\mathcal{F}\}$ first. For example, given group-by candidate $\phi(Out1, Out2) = \{shipmode, supp\}$ and the $Out5$'s aggregate value for group-by tuple `(ship, 38)` is equal to $\Lambda = 2$, the count value for its partitioned base table violates the aggregation constraint for `COUNT` aggregate. Thus, the

Table 3: Possible aggregations for every column in query output table. Blue corresponds to aggregations pruned by group-level single column constraints, and red indicates pruning by multi-column constraints. Black is for no pruning. (Aggregations pruned by table-level single-column constraints are not shown.)

λ	Set of possible aggregations $\{\mathcal{F}\}$
$Out1$	MAX($shipmode$), MIN($shipmode$)
$Out2$	MAX($supp$), MIN($supp$), AVG($supp$), SUM($supp$)
$Out3$	MAX($discount$), MIN($discount$), AVG($discount$), SUM($discount$), MAX(tax), MIN(tax), AVG(tax), SUM(tax)
$Out4$	AVG($discount$), SUM($discount$), MAX(tax), MIN(tax), AVG(tax), SUM(tax)
$Out5$	COUNT(*)

group-by candidate is invalid since there is only one possible aggregation for $Out5$ in Table 3.

In order to distinguish the invalid aggregations pruned by the group-level aggregation constraint, we label them as blue in Table 3. Take note that all the pruned aggregations correspond to the group-by candidate $shipmode$ since other group-by candidates will be pruned in this phase. For example, each aggregate value in the tuple `(air, 23, 0.00, 0.06, 1)` will be tested against the possible aggregations in Table 3 by its base table partition named `air`. As a result, those possible aggregations associated with $A = tax$ are pruned in the set $\mathcal{F}$ for $\lambda = Out3$ because they are unable to compute the aggregate value $\Lambda = 0.00$.

4.2.3 Multi-column Aggregation Constraints

Since the cost of computing group-level aggregation constraints across multiple base table partitions is significant, we can apply some additional constraints over multiple query output table columns without the involvement of base table partitions [12]. By comparing two or more query output table columns that are similarly mapped to a base attribute A, we identify the constraint rules to validate the possible dependency between them through assessing each tuple $t \in Out$ as in Table 4. For instance, given tuple `(ship, 38, 0.04, 0.05, 2)` and the base table partition named `ship`, the remaining possible aggregation for output column $Out3$ is only MAX($discount$) while the others are pruned by the group-level aggregation constraints. Therefore, the possible aggregations like AVG($discount$) and SUM($discount$) in $Out4$ can be pruned by multi-column aggregation constraints; they are labelled as red in Table 3 because the λ_{Out3} and λ_{Out4} do not satisfy any constraint rules in Table 4.

4.3 Phase 3: Filter Discovery

Once we have discovered the sets of candidates for both group-bys and aggregations, we still need to find out all the possible sets of selection predicates that can be used as the `WHERE` clause in the to-be-generated query $\mathcal{Q}$.

Filter. In our work, we define a conjunction of selection predicates, $\mathcal{C} = \mathcal{C}_1 \wedge ... \wedge \mathcal{C}_\ell$ as a filter. In order to generate the possible filters, we work backward from each aggregate value Λ in the query output table Out to its relevant base table partition by constructing its contained tuples into a matrix structure where each matrix dimension is a base attribute $A \in \mathcal{A}$. The filter is viewed as a unique bounding box in an $\mathcal{N}$-dimensional matrix over the selected $\mathcal{N}$ base attribute(s), such that the tuples inside the box are adequate to compute all the aggregate values in the query output table Out. Each edge of this unique bounding box can be expressed as a range or equality predicate in the form of $A\ op\ X$ where op is a comparison operator and X is an attribute value since each edge represents a base attribute A.

Table 4: Multi-column aggregation constraints

AGG	Constraint rules
MIN, MAX	If λ_1 is MIN(A), λ_2 is MAX(A), then $\forall t(\Lambda_1, \Lambda_2) \in Out, \Lambda_1 \leq \Lambda_2$
MIN, AVG	If λ_1 is MIN(A), λ_2 is AVG(A), then $\forall t(\Lambda_1, \Lambda_2) \in Out, \Lambda_1 \leq \Lambda_2$
MAX, AVG	If λ_1 is MAX(A), λ_2 is AVG(A), then $\forall t(\Lambda_1, \Lambda_2) \in Out, \Lambda_1 \geq \Lambda_2$
SUM, AVG	If λ_1 is SUM(A), λ_2 is AVG(A), then $\forall t(\Lambda_1, \Lambda_2) \in Out$, (1) $\lvert\Lambda_1\rvert \geq \lvert\Lambda_2\rvert$ and (2) SIGN(Λ_1) = SIGN(Λ_2)
SUM, AVG, COUNT	If λ_1 is SUM(A), λ_2 is AVG(A), λ_3 is COUNT(*), then $\forall t(\Lambda_1, \Lambda_2, \Lambda_3) \in Out, \Lambda_1 = \Lambda_2 * \Lambda_3$

Matrix. The tuples that are grouped in a base table partition can be viewed as distinct points in the matrix. For example, to construct a 2-dimensional matrix as shown in Figure 4, the base attributes $supp$ and $discount$ act as the dimensions and the three tuples in base table partition fob are positioned according to their $supp$ and $discount$ values.

Bounding Box. Given an individual matrix, the goal is to find all the possible subspaces in the matrix such that $AGG(A) = \Lambda$ where AGG is a candidate aggregation. One matrix can generate multiple bounding boxes, either overlapped or non-overlapped. These matrix subspaces are called as fuzzy bounding boxes since a fuzzy bounding box Ω is divided into inner bounding box Ω_{inner} and outer bounding box Ω_{outer} where the inner one is a subset of the outer one, $\Omega_{inner} \subseteq \Omega_{outer}$. The inner bounding box can yield a minimal matrix subspace for an aggregate value while the outer bounding box will find a maximal matrix subspace for an aggregate value. To efficiently discover all fuzzy bounding boxes, we design two different search algorithms for data exploration in the matrix.

Filter Discovery. To generate the filter with the lowest complexity, we start our filter discovery by building the individual matrices with $\mathcal{N} = 1$ for the dimensionality. After testing with all the possible base attributes $\mathcal{A}$, we increase the filter dimensionality to $\mathcal{N}+1$ if no filter is found within the current $\mathcal{N}$.

4.3.1 Expansion Search Algorithm

Given an individual $\mathcal{N}$-dimensional matrix that is built from a single base table partition, the expansion search algorithm starts from a single point, and it is expanded into $2\mathcal{N}$ directions as initial bounding boxes. For every initial bounding box, we will iteratively expand it as long as it is within the feasible region. Different types of aggregations may have different definitions for their feasible regions according to the aggregation constraints. For example, the feasible region for `MAX` aggregate cannot contain a tuple whose value is greater than the aggregate value whereas the feasible region for `SUM` aggregate requires the total sum in the bounding box should not be greater than the aggregate value. For any bounding box that moves from infeasible to feasible, it is called as inner bounding box. Conversely, it is called as outer bounding box when the bounding box moves from feasible to infeasible again. In expansion search, we discover the inner bounding box first before the outer bounding box.

Aggregations such as `MIN` and `MAX` benefit more from applying the expansion search algorithm than others. This is because they have the benefit of starting point assignment since the individual matrix contains the tuple(s) whose value is equal to the particular aggregate value. Besides, the starting point can be viewed as the inner bounding box directly and be expanded until the outer bounding boxes are found. On the contrary, it is costly for other aggregations like `COUNT`, `SUM` and `AVG`, as the expansion search has to be tried for all data points in order to find all fuzzy bounding boxes. The expansion algorithm is shown in Algorithm 2.

4.3.2 Shrinking Search Algorithm

In contrast to the approach of expansion, the shrinking search algorithm provides an alternative for finding the fuzzy bounding

Algorithm 2: Expansion

input : $\mathcal{N}$-dimensional matrix, $\mathcal{M}$ of size $|\alpha_1| \times ... \times |\alpha_\mathcal{N}|$
output: A set of fuzzy bounding boxes, $\{\Omega\}$
Initialize $\Omega_\star$ *as selected point* $p_{\alpha'_1,...,\alpha'_\mathcal{N}}$
if $\Omega_\star \subseteq$ *feasible region* **then**
  UPDATE(Ω_{inner})
foreach $\alpha_d \in \alpha_1, ..., \alpha_\mathcal{N}$ **do**
  $\Omega_{p_{\alpha'_d+1}} \leftarrow$ expand $p_{\alpha'_d+1}$ on $\Omega_\star$
  if $\Omega_{p_{\alpha'_d+1}} \subseteq$ *feasible region* **then**
    UPDATE(Ω_{inner})
  while $\Omega_{p_{\alpha'_d+1}} \subseteq$ *feasible region* **do**
    foreach $\alpha_{d'} \in \alpha_1, ..., \alpha_{d-1}$ **do**
      expand $p_{\alpha'_{d'}+1}$ and $p_{\alpha'_{d'}-1}$ on $\Omega_{p_{\alpha'_d+1}}$ iteratively
    expand $p_{\alpha'_d+1}$ on $\Omega_{p_{\alpha_d+1}}$ iteratively
    UPDATE(Ω_{outer})
  $\Omega_{p_{\alpha'_d-1}} \leftarrow$ expand $p_{\alpha'_d-1}$ on $\Omega_\star$
  if $\Omega_{p_{\alpha'_d-1}} \subseteq$ *feasible region* **then**
    UPDATE(Ω_{inner})
  while $\Omega_{p_{\alpha'_d-1}} \subseteq$ *feasible region* **do**
    foreach $\alpha_{d'} \in \alpha_1, ..., \alpha_d$ **do**
      expand $p_{\alpha'_{d'}+1}$ and $p_{\alpha'_{d'}-1}$ on $\Omega_{p_{\alpha'_d-1}}$ iteratively
    UPDATE(Ω_{outer})
  $\{\Omega\} \leftarrow$ MERGE($\Omega_{inner}, \Omega_{outer}$)

Algorithm 3: Shrinking

input : $\mathcal{N}$-dimensional matrix, $\mathcal{M}$ of size $|\alpha_1| \times ... \times |\alpha_\mathcal{N}|$
output: A set of bounding boxes, $\{\Omega\}$
Initialize $\Omega_\star$ *as given whole space* $(p_{min(\alpha_1,...,\alpha_\mathcal{N})}, p_{max(\alpha_1,...,\alpha_\mathcal{N})})$
if $\Omega_\star \subseteq$ *feasible region* **then**
  UPDATE(Ω_{outer})
foreach $\alpha_d \in \alpha_1, ..., \alpha_\mathcal{N}$ **do**
  $\Omega_{p_{max(\alpha_d)-1}} \leftarrow$ shrink $p_{max(\alpha_d)-1}$ on $\Omega_\star$
  if $\Omega_\star \subseteq$ *feasible region* **then**
    UPDATE(Ω_{outer})
  while $\Omega_{p_{max(\alpha_d)-1}} \neq$ *infeasible region* **do**
    if $\Omega_{p_{max(\alpha_d)-1}} \subseteq$ *feasible region* **then**
      UPDATE(Ω_{outer})
    foreach $\alpha_{d'} \in \alpha_1, ..., \alpha_{d-1}$ **do**
      shrink $p_{min(\alpha_{d'})+1}$ and $p_{max(\alpha_{d'})-1}$ on $\Omega_{p_{max(\alpha_d)-1}}$ iteratively
    shrink $p_{max(\alpha_d)-1}$ on $\Omega_{p_{max(\alpha_d)-1}}$ iteratively
  $\Omega_{p_{min(\alpha_d)+1}} \leftarrow$ shrink $p_{min(\alpha_d)+1}$ on $\Omega_\star$
  if $\Omega_\star \subseteq$ *feasible region* **then**
    UPDATE(Ω_{outer})
  while $\Omega_{p_{min(\alpha_d)+1}} \neq$ *infeasible region* **do**
    if $\Omega_{p_{min(\alpha_d)+1}} \subseteq$ *feasible region* **then**
      UPDATE(Ω_{outer})
    foreach $\alpha_{d'} \in \alpha_1, ..., \alpha_d$ **do**
      shrink $p_{min(\alpha_{d'})+1}$ and $p_{max(\alpha_{d'})-1}$ on $\Omega_{p_{min(\alpha_d)+1}}$ iteratively
  $\{\Omega\} \leftarrow$ MERGE($\Omega_{inner}, \Omega_{outer}$)

boxes. Given the same individual $\mathcal{N}$-dimensional matrix as expansion search, instead of creating the initial bounding boxes from the starting point(s), the edges of the matrix are shrunk from $2\mathcal{N}$ directions separately to generate the initial bounding boxes. Any initial bounding box that first moves into the feasible region is assigned as the outer bounding box and it is eligible to be shrunk before the infeasible region is reached, while the bounding box that is before the infeasible region is assigned as the inner bounding box.

Differently from the expansion search, the shrinking search is very useful for `COUNT`, `SUM` and `AVG` aggregates. This is because the fuzzy bounding boxes can be found faster once the feasible regions are reached. Hence, the cost can be reduced significantly. The algorithm for shrinking search is shown in Algorithm 3.

4.3.3 Filter Generation and Cross Validation

Given multiple fuzzy bounding boxes, one for each of the individual matrices w.r.t. the candidate aggregations, the next step is to integrate them as a unique bounding box.

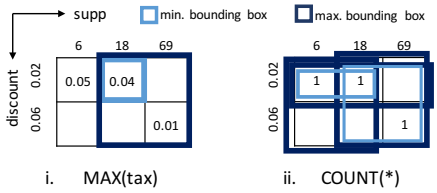
Filter Generation. As our goal is to discover a unique bounding box for the entire query, the naive approach is to take the intersection of the fuzzy bounding boxes, $\Omega_{final} = \Omega_1 \cap ... \cap \Omega_{tot}$ where tot is the total number of fuzzy bounding boxes for all individual matrices w.r.t. candidate aggregations. The cross-group intersection is taken among all the individual matrices whereas the cross-aggregation intersection takes place for all the candidate aggregations within an individual matrix. To intersect two overlapped fuzzy bounding boxes, the original inner boxes are merged through union into a larger inner bounding box while the original outer boxes are merged through intersection into a smaller outer bounding box with the condition that the inner bounding box is still contained within the outer bounding box. However, the cost of intersecting fuzzy bounding boxes will significantly increase as their number increases.

Cross Validation. One of the disadvantages of the filter generation approach is that the fuzzy bounding boxes in each individual matrix have to be generated first before the intersection takes place. To reduce such generation cost, we propose the cross validation method where we embed the fuzzy bounding boxes from one individual matrix into another individual matrix through refinement. A fuzzy bounding box can be validated if it does not show any conflict when it is embedded into a different individual matrix. However, the fuzzy bounding box can be refined by expanding the original inner box while shrinking the original outer box with the condition that the resulting inner box should not go beyond the boundaries of the resulting outer box. If the fuzzy bounding box does not match for a different individual matrix, it can be eliminated. The fuzzy bounding box that is sustained throughout all the individual matrices w.r.t. the candidate aggregations can be used to infer the resulting filter. We choose the resulting filter to be the one represented by the outer box, being the least constraining filter in the fuzzy bounding box.

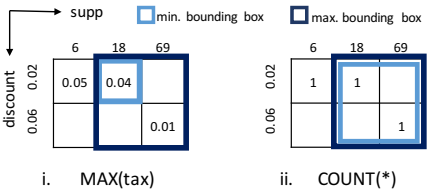
Example. Figure 4 shows the filter discovery example for base table partition fob. The matrices on the left are used to find the possible fuzzy bounding boxes for aggregation $\texttt{MAX}(tax)$ by using expansion search algorithm where the minimum bounding box is the point with value `0.04`. While on the other hand, the matrices on the right are using the shrinking search algorithm to find the possible fuzzy bounding boxes for the aggregation $\texttt{COUNT}(*)$. We show the filter generation approach in Figure 4(a) and the cross validation approach in Figure 4(b). The matrix for aggregation $\texttt{COUNT}(*)$ in Figure 4(a) yields three fuzzy bounding boxes when using filter generation approach. However, this cost can be reduced when using cross validation approach, where the matrix for aggregation $\texttt{COUNT}(*)$ yields one fuzzy bounding box as shown in Figure 4(b).

4.4 Efficiency and Discussion

As we previously mentioned, existing reverse engineering approaches [8, 9, 15] that handle group-by and aggregation queries have no guarantee that they output the query with the least complexity. The decision tree based method could generate complex queries with a large number of selection predicates. Conversely, our approach is able to generate the queries in order of increasing complexity according to a user defined complexity function (e.g., number of group-by attributes, number of selection predicates etc.)



(a) Filter Generation Approach



(b) Cross Validation Approach

Figure 4: A filter discovery example for base table partition fob

if multiple feasible queries exist, or terminate the discovery process earlier if certain criteria are met (e.g., the simplest query is found, a complexity threshold is reached).

To analyse the time complexity of our algorithm, let T, t denote the number of tuples in the base table and query output table respectively, and K, k denote the number of columns in the base table and query output table respectively. In Phase 1, the number of mappings is bounded by K^k; for each possible mapping, the size of the lattice is bounded by 2^k; keyness checking for a lattice node can be done using sorting in time $O(t*log(t))$; and similarly containment checking can be done in time $O(t*log(t)+T*log(T))$. Thus the total time complexity of Phase 1 is $O(K^k * 2^k * (t*log(t)+T*log(T)))$. In Phase 2, inverted indexes need to be built for the base table, partitioned according to the choice of grouping attributes, which has a time cost of $O(K*T)$. There are $O(K^k)$ mappings of aggregate columns; for each mapping, the table-level single-column and multi-column aggregation constraints can be checked in time $O(k*(t+T))$; and the group-level aggregation constraints can be checked in time $O(k*t*T)$. Thus, the total complexity of Phase 2 is $O(K^k*(K*T+k*t*T))$. In Phase 3, let V denote the maximum number of distinct values of any base table column; since a fuzzy bounding box is identified by at most 4 values in each dimension, the number of bounding boxes in a matrix of dimensionality d is bounded by $O(V^{4*d})$; also, each bounding box needs to include a distinct subset of tuples from the base table, providing another upper bound of $O(2^T)$ for the number of bounding boxes. Since both expansion search and shrinking search explore the space of possible bounding boxes, the overall time complexity of Phase 3 is $O(k*t*min(2^T, V^{4*K}))$. The total cost of the algorithm is the sum of the costs of the three phrases. The time complexity is analyzed based on worst-case scenario. As we show in the experiment section, the proposed optimizations in each phase provide substantial pruning in practice.

5. EXPERIMENTAL STUDY

We empirically evaluate the performance of our proposed algorithm and discuss the experimental results. We first compare the effectiveness of our three-phase algorithm called REGAL with QRE [15]. Next, we look at the performance of each individual phase in the algorithm, and then examine the overall algorithm performance based on our considerations in each individual phase.

Implementations. We use the TPC-H benchmark dataset with scale factor 1, which is 1GB in size. All the algorithms were coded in Java and available at Github [1]. The experiments were conducted on an Ubuntu Linux 12.04 machine with an Intel Xeon 2.40GHz CPU and 32GB RAM running MySQL. Each experiment is run three times and we report the average timing over them.

Test Queries. In order to study the performance of different phases in our algorithm, we generate the list of SQL queries shown below where the group-by clause is abbreviated as $\mathcal{G}$ and $\mathcal{F}_{ALL}$ denotes the 5 basic aggregate operators. Symbols σ_{option}, $\sigma_{selectivity}$, and $\sigma_{dimensionality}$ represent the variations of selection predicates in the particular test queries whereas $\pi_{aggregation}$ and π_{group} represent the variations of aggregations and group-bys in the particular test queries.

$$\mathcal{Q}_1 = \pi_{brand,\mathcal{F}_{ALL}(extendedprice)}\mathcal{G}_{brand}\ \sigma_{tax>0.04}(lineitem \bowtie part)$$

$$\mathcal{Q}_2 = \pi_{suppkey,discount,MAX(orderkey),MAX(extendedprice)}\ \mathcal{G}_{suppkey,discount}(lineitem)$$

$$\mathcal{Q}_3 = \pi_{suppkey,MAX(extendedprice),\mathcal{F}_{ALL}(totalprice)}\ \mathcal{G}_{suppkey}\sigma_{option}(lineitem \bowtie orders)$$

$$\mathcal{Q}_4 = \pi_{shippriority,\mathcal{F}_{ALL}(extendedprice)}\mathcal{G}_{shippriority}\ \sigma_{selectivity}(lineitem \bowtie orders)$$

$$\mathcal{Q}_5 = \pi_{linenumber,\mathcal{F}_{ALL}(extendedprice)}\mathcal{G}_{linenumber}\ \sigma_{suppkey>5000 \wedge suppkey<9990}(lineitem)$$

$$\mathcal{Q}_6 = \pi_{linenumber,\mathcal{F}_{ALL}(extendedprice)}\mathcal{G}_{linenumber}\ \sigma_{size>24 \wedge size<48}(lineitem \bowtie part)$$

$$\mathcal{Q}_7 = \pi_{shipmode,SUM(quantity)}\mathcal{G}_{shipmode}\ \sigma_{dimensionality}(lineitem)$$

$$\mathcal{Q}_8 = \pi_{linenumber,tax,SUM(quantity)}\mathcal{G}_{linenumber,tax}\ \sigma_{quantity>10}(lineitem)$$

$$\mathcal{Q}_9 = \pi_{linestatus,aggregation}\mathcal{G}_{linestatus}\ \sigma_{tax>0.01 \wedge tax<0.04 \wedge discount>0.03}(lineitem)$$

$$\mathcal{Q}_{10} = \pi_{group,MAX(extendedprice)}\mathcal{G}_{group}\ \sigma_{quantity>9 \wedge quantity<49}(lineitem)$$

$$\mathcal{Q}_{11} = \pi_{linenumber,AVG(extendedprice)}\mathcal{G}_{linenumber}\ \sigma_{tax>0.04}(lineitem)$$

$$\mathcal{Q}_{12} = \pi_{linenumber,MEDIAN(extendedprice)}\mathcal{G}_{linenumber}\ \sigma_{tax>0.04}(lineitem)$$

5.1 Comparison with QRE

We compare our proposed 3-phase algorithm against QRE [15] by adopting their solution to run the experiment. In QRE default settings, the query output table Out consists of 2 projected columns that are joined from different relations and one column is assigned for group-by clause whereas another column is assigned as an aggregate. Therefore, to compare with QRE, we run test query $\mathcal{Q}_1$ against all basic aggregate operators. Based on the experiment result in Figure 5, we can see that our proposed algorithm consistently outperforms QRE. This is because our algorithm is more aggressive in its pruning strategies and starts by finding possible filter predicates from a single dimension. Then, it only increases dimensionality as needed until the query with the least complexity is found, which is different from the TALOS approach of classifying tuples by using decision trees.

5.2 Group-by Discovery

Figure 6 shows the performance of different lattice traversal strategies for given test query $\mathcal{Q}_2$ in the phase of group-by discovery. The group-by discovery has two main methods of checking, so we start the lattice traversal with either keyness or containment checking, in either top-down or bottom-up fashion. We assess both the running times and the number of checked nodes for the experiment. The running time of keyness checking is generally smaller than that of containment checking, because the cost of running keyness checking only involves the query output table which is rather small while the cost of running containment checking is more since it involves scanning both base and query output tables. Thus, it is better

that keyness checking is started before containment checking in order to reduce the number of checked nodes in containment checking. Figure 6(a) shows that the running time of lattice traversal via bottom-up keyness and top-down containment is the smallest as compared with other alternatives even though it has checked more nodes than other alternatives which can be shown in Figure 6(b).

5.3 Grouping and Aggregates Pruning

In order to assess the effectiveness of our pruning strategy, we use test query $\mathcal{Q}_3$ for this experiment. We use all five basic aggregate operators for attribute `totalprice` and combine each of them with an invariant aggregation called `MAX` (`extendedprice`). Both attributes `extendedprice` and `totalprice` have the same data type, and the range of attribute `extendedprice`, [901.0, 104949.5], is a subset of the range of attribute `totalprice`. Further, we alter the test query by putting an optional filter with either (i) null, (ii) non-grouping attributes, or (iii) both grouping and non-grouping attributes.

Generally, based on the performance in Figure 7, the time taken by pruning phase is mainly influenced by group-level aggregation constraints. This is because the base table needs to be scanned once for table-level aggregation constraints which takes less than one second. The test query $\mathcal{Q}_3$ with filter option (iii) takes the smallest running time. This is because half the groups are filtered out, and hence the number of base table partitions used for group-level aggregation constraints is also halved. On the other hand, $\mathcal{Q}_3$ with filter option (i), or no filter, takes smaller time than filter option (ii). This is because the aggregation candidates are increased when there is a filter on a non-grouping attribute applied. For example, the test query $\mathcal{Q}_3$ with `MIN` (`totalprice`) and filter option (ii), both `MAX` (`extendedprice`) and `MIN` (`totalprice`) match to six candidates by table-level aggregation constraints, and then two out of three group-by candidates are pruned by group-level aggregation constraints where the remaining group-by candidate `suppkey` will return totally 26 out of 36 combinations of aggregations for filter discovery to find the possible selection predicates. Meanwhile, in the absence of the pruning phase altogether, for the test query above, both `MAX` (`extendedprice`) and `MIN` (`totalprice`) need to test for all possible candidates, which each contributes eight candidates, i.e. `MAX/MIN/SUM/AVG` (`extendedprice`) and `MAX/MIN/SUM/AVG` (`totalprice`). This ends up with 64 combinations of aggregations from their cross product for filter discovery, which would significantly impact the running time of the overall algorithm by three times or higher. However, regardless of the filter option, aggregations such as `AVG`, `SUM` and `COUNT` need smaller running time compared to `MAX` and `MIN`. This is because fewer candidates can be generated as their aggregation constraints are restricted by the number of tuples involved. For instance, `SUM` (`totalprice`) can match to two candidates, namely `SUM` (`extendedprice`) and itself.

5.4 Expansion Search vs. Shrinking Search

Figure 8 plots the running time of filter discovery from a single one-dimensional individual matrix for test query $\mathcal{Q}_4$ with all proposed aggregate operators. In order to distinguish the effectiveness of our search algorithms, we vary the selectivity of the filter to be discovered. The filter selectivity will impact the feasible region in a matrix to be selected as a filter. The higher the selectivity value, the larger the feasible region which implies there are huge amount of tuples contained in the filter. In the experiment, we represent a selectivity by applying a specific range predicate on attribute `suppkey` which contains 10000 distinct values. As Figure 8 shows, for the aggregations with `MAX` and `MIN`, the expansion search algorithm is found useful especially when the selectivity is extremely low. In contrast, the shrinking search algorithm is more effective for the aggregations with `COUNT`, `SUM` and `AVG` especially when the selectivity value is extremely high. Since aggregation for `AVG` needs division, it runs slower than other aggregations.

5.5 Sparsity of an Individual Matrix

Figure 9 shows the running time of filter discovery for individual matrices in different levels of sparsity against test query $\mathcal{Q}_5$. The sparsity of an individual matrix indicates the number of tuples contained in the individual matrix where a sparser individual matrix would have fewer tuples. Thus, given the test query $\mathcal{Q}_5$, we select three base partitions with approximately 0.5M, 1.0M and 1.5M tuples respectively to run the experiment. From the experiment result, it is clearly seen that the running time is smaller when the individual matrix contains fewer tuples. This is because the individual matrix will be constructed in a shorter time with fewer feasible regions found as the matrix is sparser.

5.6 Filter Generation vs. Cross Validation

In order to integrate all the bounding boxes of all individual matrices to discover the filter, we have proposed two main methods which are known as filter generation and cross validation. To assess their effectiveness, we run these two methods for the test query $\mathcal{Q}_6$ by varying both the number of individual matrices used for filter generation and cross validation to discover the filter. Furthermore, as the sparsity of an individual matrix will also affect the running time of finding fuzzy bounding boxes, we run the experiment for test query $\mathcal{Q}_6$ that is grouped by attribute `linenumber` where the base partitions have a skewed distribution. To better understand the performance, we run the experiment in both ascending and descending orders of the base partition size. However, Figure 10 shows the ascending case only as we have found that processing the groups in an ascending order of their sizes was consistently faster than the descending order on average by 13% for all the cases in the figure. Hence, we can conclude that keeping the smallest base table partition for filter generation while leaving other larger base table partitions for cross validation would be the most time-efficient.

5.7 Filter Dimensionality

Figure 11 depicts the running time of filter discovery for the test query $\mathcal{Q}_7$ while the filter dimensionality is varied from $\mathcal{N} = 1$ to $\mathcal{N} = 3$. Table 5 shows the different selection predicates used as filters for the test query $\mathcal{Q}_7$. Generally, the running time increases significantly as the filter dimensionality is increased. Even for the filters with similar dimensionality, such as $\mathcal{S}_4$, $\mathcal{S}_5$ and $\mathcal{S}_6$ with $\mathcal{N} = 2$ in the experiment, their running time results are different due to the cost of constructing and searching fuzzy bounding boxes in the individual matrices. As the attributes used for filters have more domain values, their individual matrices will be larger and thus more time is needed to explore them.

5.8 Number of Lattices

In TPC-H dataset, there are some attributes that share common values. For instance, the attribute `Linenumber (L)` has common values with the attributes `Suppkey (S)`, `Partkey (P)`,

Table 5: Query $\mathcal{Q}_7$ with different selection predicates

	Filter	$\mathcal{N}$	Sel
$\mathcal{S}_1$	$linenumber \leq 5$	1	0.89
$\mathcal{S}_2$	$tax \geq 0.01$	1	0.89
$\mathcal{S}_3$	$quantity < 46$	1	0.89
$\mathcal{S}_4$	$linenumber \leq 5 \wedge tax \geq 0.01$	2	0.79
$\mathcal{S}_5$	$linenumber \leq 5 \wedge quantity < 46$	2	0.80
$\mathcal{S}_6$	$tax \geq 0.01 \wedge quantity < 46$	2	0.80
$\mathcal{S}_7$	$linenumber \leq 5 \wedge tax \geq 0.01 \wedge quantity < 46$	3	0.71

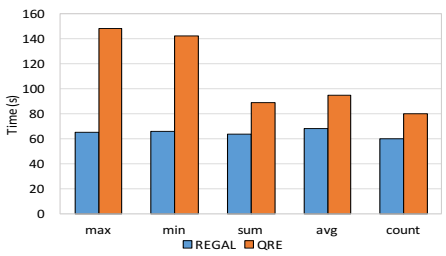
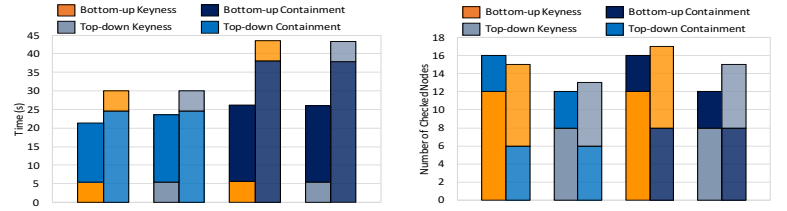


Figure 5: Comparison with QRE



(a) Running Time **(b)** Number of Checked Nodes

Figure 6: Effect on lattice traversal strategy

and `Orderkey (O)` whereas the distinct values of `Tax (T)` overlap with `Discount (D)`. Therefore, multiple lattices will be generated in the case of many-to-many mappings. In test query $\mathcal{Q}_8$, we evaluate the overall algorithm by controlling the number of lattices due to the mappings and the experiment result is shown in Figure 12. As the number of lattices increases, the possible group-by and aggregation candidates also increase, thus additional running time is needed to evaluate both phase 1 and phase 2. Besides, the lattices may cause an increase in possible combinations of aggregations in phase 2 where each of them is needed for filter discovery. However, it is not that every combination will produce a valid filter as some combinations are invalid candidates. Therefore, we can restrict the attributes used for filter generation as well as set the threshold such that the dimensionality of discovered filters is at most three, $\mathcal{N} \leq 3$. Then, filter discovery for a false positive combination of aggregations will be aborted when it doesn't have any filter that can be generated.

5.9 Precision and Recall

In order to investigate the accuracy of our algorithm, we measure the precision and recall for the test queries ($\mathcal{Q}_1$-$\mathcal{Q}_8$) which we had used in the experiments. Each test query is treated as true query while any returned query candidate which is able to discover the true query is considered as true candidate. Specifically, the true candidate has the same aggregation combination and filter complexity as the true query, and its fuzzy bounding box can subsume the original bounding box in the true query. If the true candidate was found, the recall is given as 1 whereas the precision is calculated as the ratio of true candidate over all the returned query candidates. Table 6 records the precision and recall for all the test queries. Since all queries $\mathcal{Q}_{1-8}$ except $\mathcal{Q}_4$ contain enough information in their query output table for the algorithm to find the true candidate, both precision and recall are 1 because only the true candidate is returned. On the other hand, $\mathcal{Q}_4$ only provides one tuple and two columns (one group-by, one aggregation), and the aggregation is varied by $\mathcal{F}_{ALL}$. For some aggregations, $\mathcal{Q}_4$ introduces some ambiguity as the algorithm could find many possible query candidates that produce the exact same query output table as $\mathcal{Q}_4$. In the experiment, we list the parameter $\mathcal{F}_{ALL}$ of $\mathcal{Q}_4$ in different variations while we put it as $*$ for other queries as their results in different variations are the same. Besides the parameter $\mathcal{F}_{ALL}$, we set the other parameters in their default settings, e.g. we set the $\sigma_{selectivity}$ in $\mathcal{Q}_4$ as 0.5 with the range predicate `suppkey` in the interval between 3000 and 8000. By looking at $\mathcal{Q}_4$, aggregations such as `sum` and `avg` can yield the specific values, hence the

Table 6: Precision and recall for all test queries

Query	$\mathcal{F}_{ALL}$	#rows (min-max)	#cols (min-max)	Average precision	Average recall
$\mathcal{Q}_4$	`min/max`	1	2	1/20	1
	`sum/avg`	1	2	1	1
	`count`	1	2	≈ 0	1
$\mathcal{Q}_{1-3,5-8}$	$*$	7-110,000	2-4	1	1

Table 7: Precision and recall for Query $\mathcal{Q}_9$

Aggregation	#cols	Precision	Recall	k
`MAX(`$extendedprice$`)`	2	0/4	0	6
`MAX(`$extendedprice$`),COUNT(`$*$`)`	3	1	1	1
`MAX(`$extendedprice$`),COUNT(`$*$`),` `SUM(`$extendedprice$`)`	4	1	1	1

precision and recall for both are 1. On the other hand, when aggregations such as `min`, `max` and `count` are used, there can be many selection predicates with the same filter complexity as $\mathcal{Q}_4$ that provide the same output tuple as $\mathcal{Q}_4$. The precision deteriorates as the number of returned query candidates increases significantly, which is especially true for `count`, where the precision is almost zero.

To further analyse this problem, we added $\mathcal{Q}_9$ and $\mathcal{Q}_{10}$ to investigate whether the increasing the number of columns in $\mathcal{Q}_9$ and the number of rows/groups in $\mathcal{Q}_{10}$ will increase the precision and recall. The experiment results are reported in Table 7 for $\mathcal{Q}_9$ and Table 8 for $\mathcal{Q}_{10}$. As shown in Table 7, when the query output table only contains two columns, both precision and recall are zero because all four returned candidates are in the lowest filter complexity (one-dimensional) while the true query itself has the filter complexity of two (`tax` and `discount` are used as selection predicates). In order to guarantee the true candidate is returned, the least number of candidates that are needed to be returned is k= 6. On the other hand, when the number of columns is increased by adding more aggregations, many invalid candidates will be pruned as only the true candidate is returned, therefore both precision and recall are 1. For the test query $\mathcal{Q}_{10}$, we control the number of tuples in the query output table in order to measure the precision and recall. A tuple in the query output table is actually same as a group, hence adding an extra group-by attribute in the true query is actually increasing the number of groups. The test query $\mathcal{Q}_{10}$ with $\mathcal{G}_{linestatus}$ will output two distinct rows whereas $\mathcal{Q}_{10}$ with $\mathcal{G}_{linestatus,returnflag}$ will output four distinct rows. Table 8 shows that as the number of rows in the query output table is increased, the precision is also increased since the number of candidates is fewer and the recall is always 1 since the true candidate is discovered.

5.10 Scalability

We use the scale factors, namely 0.01, 0.1 and 1 in logarithm scale to represent the increase of TPC-H data size. As we use all the test queries and since different queries take different times, we normalize the time spent for each test query in scale factor 0.01 as 1 time-unit for other scales. After running all queries, we plot the average and standard deviation for all other scales (i.e. 0.1 and 1). Figure 13 shows that the execution time against the scalability of dataset increases linearly in a log-log plot.

Table 8: Precision and recall for Query $\mathcal{Q}_{10}$

Group	#rows	Precision	Recall
$linestatus$	2	1/4	1
$linestatus, returnflag$	4	1	1

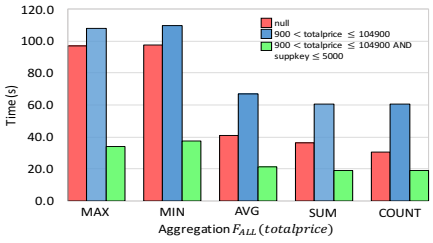


Figure 7: Effect of pruning strategy

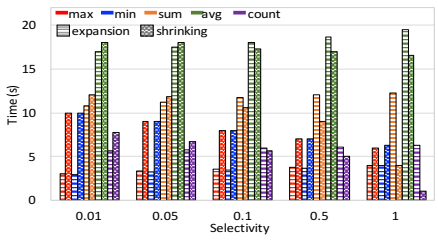


Figure 8: Expansion vs. shrinking

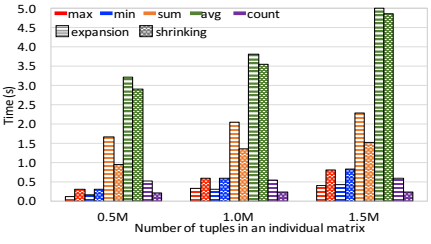


Figure 9: Effect of matrix sparsity

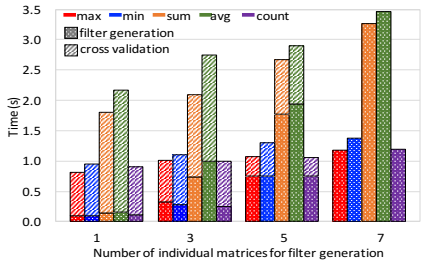


Figure 10: Filter generation vs. cross validation

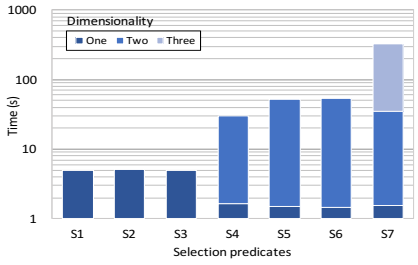


Figure 11: Effect of filter dimensionality

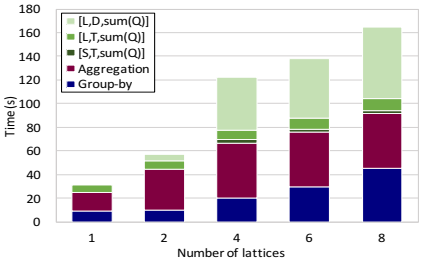


Figure 12: Effect of number of lattices

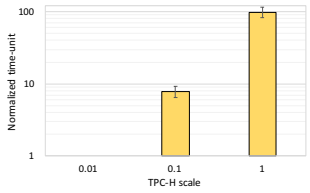


Figure 13: Scalability

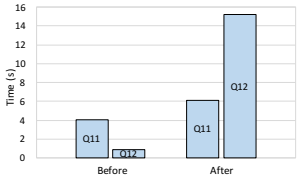


Figure 14: Median function

5.11 Extension to User-defined Functions

In our paper, we use five pre-defined basic aggregate functions to discover the possible aggregations in the query output table. In order to extend to additional user-defined functions, not only do we have to design the particular constraint rules, but also their relationships with current pre-defined constraint rules. To illustrate, we have taken the `MEDIAN` function as an example for extension in our algorithm. For the phase of grouping and aggregates pruning, the constraint rules for `MEDIAN` are similar to `AVG` since the median values always lie between the `MAX` and `MIN` and those values are not necessary the subset of the particular base table attribute values. Therefore, any candidate that is matched to `AVG` will be also matched to `MEDIAN` simultaneously without adding new pruning rules. For the phase of filter discovery, in order to find the bounding boxes in a matrix, the values inside the selected region are sorted before the `MEDIAN` is computed. The selected region is treated as a bounding box when the median of the enclosed values is the same as the input value. We use the test queries $\mathcal{Q}_{11-12}$ to explore the effect of adding `MEDIAN` into our algorithm. Figure 14 plots the time spent for generating the bounding boxes before and after adding the `MEDIAN` into our algorithm. Before adding `MEDIAN`, the test query $\mathcal{Q}_{12}$ returned empty result since it did not match to any selection predicate while the time spent in Phase 3 is used to eliminate the possible candidates in Phase 2. After adding `MEDIAN` in the algorithm, the time needed for $\mathcal{Q}_{12}$ is doubled as compared to $\mathcal{Q}_{11}$. This is because additional sorting in each shrinking step in needed to compute the median values as we determine the bounding boxes. Besides, the time spent in $\mathcal{Q}_{11}$ after introducing `MEDIAN` has increased since extra checks are needed in order to eliminate the `MEDIAN` possibility.

6. CONCLUSION

In this paper, we proposed a novel algorithm for reverse engineering aggregation queries. Our algorithm operates in three phases: (a) to discover group-by attributes, (b) to prune combinations of group-by and aggregate attributes, and (c) to identify the selection filters. We perform an extensive experimental study over TPC-H benchmark data to show the versatility and scalability of our algorithm.

7. REFERENCES

[1] Regal algorithm. Available at `https://github.com/weichit/Regal`.

[2] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: A survey. *The VLDB Journal*, 24(4):557–581, Aug. 2015.

[3] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: Enabling keyword search over relational databases. In *SIGMOD*, pages 627–627, 2002.

[4] A. Bonifati, R. Ciucanu, and S. Staworko. Learning join queries from user examples. *ACM Trans. Database Syst.*, 40(4):24:1–24:38, Jan. 2016.

[5] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. In *ICDT*, pages 89–103, 2010.

[6] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[7] H. Li, C.-Y. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. *PVLDB*, 8(13):2158–2169, 2015.

[8] K. Panev and S. Michel. Reverse engineering top-k database queries with paleo. EDBT, 2016.

[9] K. Panev, E. Milchevski, and S. Michel. Computing similar entity rankings via reverse engineering of top-k database queries. In *ICDEW*, pages 181–188, 2016.

[10] J. Picado, A. Termehchy, and A. Fern. Schema independent relational learning. *CoRR*, abs/1508.03846, 2015.

[11] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD*, pages 73–84, 2012.

[12] K. A. Ross, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Foundations of aggregation constraints. *Theor. Comput. Sci.*, 193(1-2):149–179, 1998.

[13] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. In *SIGMOD*, pages 493–504, 2014.

[14] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD*, pages 535–548, 2009.

[15] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query reverse engineering. *The VLDB Journal*, 23(5):721–746, Oct. 2014.

[16] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD*, pages 809–820, 2013.

[17] M. M. Zloof. Query by example. In *AFIPS NCC*, pages 431–438, 1975.