Efficient Query Reverse Engineering for Joins and OLAP-Style Aggregations

Wei Chit Tan(✉)

Singapore University of Technology and Design, Singapore, Singapore
weichit_tan@mymail.sutd.edu.sg

Abstract. Query reverse engineering is getting important in database usability since it helps users to gain technical insights about the database without any intentional knowledge such as schema and SQL. In this paper, we review some existing techniques that focus on join query discovery, and we devise our efficient algorithm to discover the SQL queries that contain both joins and OLAP-style aggregations which are substantially for querying OLAP data warehouses. We show that our algorithm is adaptable and scalable for large databases by performing an empirical study for TPC-H benchmark dataset.

1 Introduction

Since every organization may have its unique data warehouse and it is always managed and maintained by a team of technical experts, it is rather hard for ordinary users to make full use of these generated data, especially those spreadsheets from the data warehouse. For a general purpose, database users are required to learn both schema and query language, which are important for them to invoke the tuples from the relevant relations precisely. Thus, the SQL join operations are definitely important for combining the relevant columns from these tables into a common (denormalized) table. Besides, these combined data are often associated with OLAP-style aggregations (e.g., basic mathematical operators) for offering more valuable insights about the numerical data.

Figure 1 illustrates a motivating example. Figure 1(a) is an example spreadsheet table, and Fig. 1(b) shows are a pair of or even better minimal join graphs that could regenerate this spreadsheet table through different join tables, projections, and aggregations. A candidate join graph is akin to a schema graph. Each node represents a relation, and it is starred if it contains a projection column. Therefore, from the candidate join graphs, only the validated join graph would be executed for discovering other SQL classes, e.g., OLAP group-by, aggregations and selection filters.

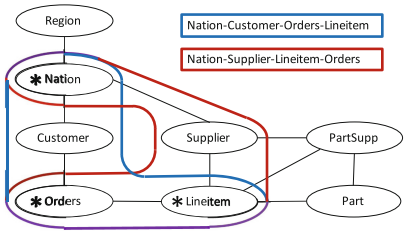
1.1 Related Work

Instead of using a keyword query that is made up of several keywords, there are many proposals have been implemented to discover join queries by using a tabular list of tuples as the implication of keyword search in relational databases [7].

Y. Cai et al. (Eds.): APWeb-WAIM 2018, LNCS 10988, pp. 53–62, 2018.
https://doi.org/10.1007/978-3-319-96893-3_5

N_NAME	L_LINESTATUS	MAX(O_TOTALPRICE)	SUM(L_QUANTITY)
ARGENTINA	O	530604.44	2284691.00
CANADA	F	510061.60	2343191.00
CANADA	O	515531.82	2284164.00
FRANCE	F	508668.52	2343432.00
IRAN	F	522644.48	2276219.00
JAPAN	O	502742.76	2287464.00
MOZAMBIQUE	O	508047.99	2348205.00
PERU	F	544089.09	2269762.00
PERU	O	522720.61	2264220.00
RUSSIA	F	555285.16	2359354.00
UNITED STATES	O	525590.57	2316886.00
VIETNAM	F	504509.06	2301689.00

(a) An example spreadsheet table



(b) Join candidate graphs

Fig. 1. The different join queries that are possible to generate an example spreadsheet.

Most of the existing solutions (e.g. [10,11,13]) depend on schema-based approach [1,2], and the database schema is illustrated as a graph by taking the relations as nodes and the foreign key references as edges. In DISCOVER [2] and its extended works [3,7], given that a set of candidate networks discovered by a keyword query, the candidate network evaluation needs an optimized execution plan which is depicted as an operator tree in order to translate each of them into SQL. Nonetheless, the full-text search is another technique to verify candidate queries by emphasizing keyword containments as SQL predicates, which it is exceptionally useful for text attributes and built-in indexes are required in advance. Several works [5,6,8] support this full-text search feature, thus the query discovery is restricted to textual databases in lieu of the OLAP data warehouses.

Another critical factor that could optimize the join execution is the indexing techniques. In lieu of joining every projection attribute for candidate query evaluation, the implementation of join indices [12] only require those relevant primary keys to form a temporal relation so that the overhead memory cost can be avoided. The well known TALOS framework [10,11] uses the join indices to build an intermediate join relation and thus applies the decision tree classifier to classify the tuples for selection predicate generation. In addition, as indicated in [13], the unique tuple identifiers (*tids*) within each relation are used to examine each schema-based connected tree at instance-level in order to invalidate any schema trees that cannot generate a random output tuple.

Apart from that, besides those fundamental SQL classes which can determine the schema tables and attributes for query discovery, other classes such as `HAVING` and `ORDER BY` clauses have their specifications to produce the finalized SQL results. PALEO framework [4] uses the concept of ranked list of tuples to reverse engineer OLAP queries where each query contains an `ORDER BY` column.

1.2 Contributions

Our contributions in this paper are presented as follows:

– We provide a solution that generates the candidate join graphs through the schema and metadata exploration to characterize each distinct column of query output table.

– We improve the expressiveness of our solution by discovering SQL `HAVING` clause for aggregation queries.
– We prove that our algorithm is adaptable and scalable by conducting an experimental evaluation over the standard TPC-H dataset.

2 Problem Definition

A relational database $\mathcal{D}$ consists of a set of relations and every relation is linked by referential integrity constraints. The relational schema is defined as a schema graph $\mathcal{SG}(\mathcal{R}, \zeta)$, where each $\mathcal{R}$ is a table and each ζ is an fk/pk constraint. A subgraph $\mathcal{J}(\mathcal{G})$ entails a join query where a relation $\mathcal{R} \in \mathcal{SG}$ may appear more than once as a node in it. The Project-Join (PJ) queries should contain at least both projection (π) and join ($\bowtie$) operations where the projection determines the number of columns and the join determines the number of relations. A subgraph $\mathcal{J}(\mathcal{G})$ connects all the relevant relations through their fk/pk constraints while it may contain other relations as well as intermediate nodes to interconnect all the relevant relations. Hence, the schema size is directly proportional to the size of $\mathcal{J}(\mathcal{G})$. To prune the overwhelming unnecessary tuples from the outputted join table, the selection operation (σ) is used as a filter by specifying the necessary conditions for the query output table Out. The formulated queries with these three SQL operations are named as Select-Project-Join (SPJ) queries. In our work, we intend to discover more complex queries than the SPJ queries, i.e., the OLAP queries. Given the query output table Out as input, the `GROUP BY` operator will correspond to the number of tuples (groups) of Out. Each group will be used to produce one or multiple aggregations where each aggregation takes an aggregate operator (e.g. `MAX`, `MIN`, `AVG`, `SUM`, and `COUNT`) for a numeric attribute. Upon the above OLAP specifications, we define the queries as Select-Project-Join-Aggregation (SPJA) queries.

3 Join Query Discovery

In this section, we discuss how to discover the possible subgraphs based on a given query output table Out. Its columns are essential to delimit the schema size for query regeneration. Our join query discovery relies on a graph search algorithm to determine the possible candidate subgraphs. For instance, the breadth-first search algorithm in DISCOVER [2] finds the subgraphs where the nodes that contain the given keywords are taken as the leaf nodes. Apart from just considering the keywords, our problem is to find out all the possible subgraphs that can cover all columns in Out. Algorithm 1 indicates the join query discovery.

3.1 Column Mapping Table

Consider a column of Out, it is outputted by the projection operation (π) for a schema attribute A, either is operated as group-by or aggregation. An SPJA

query, that aggregates the output tuples from the sets of grouped tuples; there are some columns whose aggregate values cannot be directly mapped to any schema attributes. Due to the possibility of unidentified/anonymous schema attribute(s), the column mapping details may be incomplete. To solve this problem, for each unmapped column, it can match a set of covering attributes; otherwise it has to be an integer column that can be corresponded to `COUNT` aggregation. These covering attributes are discovered due to different mathematical properties that are possessed by different aggregate functions. However, it is non-trivial to determine the set of covering attributes intuitively if the unmapped column tuples are far beyond any minimum/maximum values of schema attributes which the only possibility is the `SUM` computation that relates with both `COUNT` and `AVG`.

3.2 Candidate Subgraph Generation

By assuming the schema graph is undirected, the current (in)complete column mapping table is used to search for the (partial) candidate subgraphs. The mapped relations are set as leaves so that they must be contained in the candidate subgraph generation. A set of partial subgraphs is generated due to incomplete column mapping. Given a partial subgraph, it will be either explored or expanded to find the covering attribute(s) for the unmapped column(s) of column mapping table. In Fig. 1, the relations named `Nation`, `Lineitem` and `Orders` are the leaves because the schema attributes `Nation.name`, `Lineitem.linestatus`, and `Orders.totalprice` are mapped. When exploring the discovered partial subgraph, the attribute `Lineitem.quantity` can be the covering attribute for the last column of *Out*.

Partial Subgraph. Consider a set of leaves, the least connected leaf node is selected as root to connect other leaves to form a subgraph through the undirected schema graph via breadth-first search exploration. If there exists a pair of same leaf nodes, the node duplication is allowed where a node can be visited for twice. The schema size thus is determined by the total number of visited nodes. To control the schema size as well as the cost complexity, the number of intermediate nodes should be kept as fewer as possible. By heuristically, the candidates are sorted by the schema size for evaluation.

Join Table Size Estimation. Upon a partial subgraph, by doing schema exploration, the utmost task is to complete the column mapping table. Once every column in *Out* has its corresponding schema attribute(s), the partial subgraph thus becomes the complete candidate subgraph. For an unmapped column that contains aggregation results, the idea is to find the corresponding numeric attributes. Among the possible candidates, the priority is to quickly prune the inappropriate ones by inferring its join size. For a partial subgraph, its join size, Υ is determined by the total number of tuples to generate *Out*. In addition, its schema is equivalent to a set of attributes, denoted as $\mathcal{A}$. If an unmapped column λ contains only natural numbers, its total number is considered the estimated

Algorithm 1. Join Query Discovery

```
input  : SG: schema graph, Out: query output table
output: {J(G)}: set of candidate subgraphs
//Column Mapping
mapping table φ = ∅
covering table φ̄ = ∅
foreach column λ ∈ Out do
    if φ(λ) = schema attribute A then
        update φ(λ) ← A
    else
        insert λ into φ̄
//Candidate Subgraph Generation
foreach mapping φ do
    find partial subgraphs from SG
    foreach partial subgraph do
        if φ̄ ≠ ∅ and φ̄(λ) = schema attribute A then
            update φ̄(λ) ← A
            {J(G)} ← partial subgraph
        else if φ̄ ≠ ∅ and !φ̄(λ) = schema attribute A then
            find set of neighbour nodes {R}
            while expandPartialSubgraph(R) do
                if φ̄(λ) = schema attribute A then
                    update φ̄(λ) ← A
                    {J(G)} ← partial subgraph
        {J(G)} ← partial subgraph
```

join size, $\Gamma = \sum \Lambda$, where $\Lambda \in \lambda$. The estimated Γ should be within the range of $\alpha * \Upsilon$ and Υ where α is the selectivity factor that delimits the number of tuples to generate Out as the impact of applied selection conditions. We assume the default value of α as 0.1 and all data are in normal distribution. If the statement is true, then it can be delineated as the computation of `COUNT`$(*)$. However, for the implication of $SUM(A)$, given that A is a schema attribute where $A \in \mathcal{A}$ and an unmapped numerical column λ, the estimated join size, Γ can be calculated by a simple formula as below.

$$\frac{\sum_{i=1}^{|\lambda|} \Lambda_i}{AVG(A)} = \Gamma \quad \begin{cases} \alpha * \Upsilon \leq \Gamma \leq \Upsilon & \text{true} \\ \text{otherwise} & \text{false} \end{cases}$$

An attribute $A \in \mathcal{A}$ is acceptable if the estimated Γ is between $\alpha * \Upsilon$ and Υ.

Expanding Neighbour Nodes. If it still does not have any covering attribute for any unmapped column of Out, the partial subgraph cannot establish as a candidate subgraph. An alternative approach is to expand the current partial subgraph by adding one of its neighbour nodes to form a new subgraph for schema exploration. If every column of Out is being mapped or covered, the new subgraph is added to the set of candidate subgraphs; otherwise, another new subgraph is generated by adding another selective neighbour node. This process is iterated until the set of candidate subgraphs is found.

4 Group-By Discovery, Aggregates Pruning and Filter Discovery

After determining the possible joins, the next step is to determine the group-by candidates for query discovery. According to the mapping columns, a group-by lattice is built where its nodes are the group-by candidates and its edges are the superset-subset relationships. The invalid nodes are pruned by exploring the lattice, and the remaining nodes are the possible candidates for subsequent aggregates pruning. The rule-based aggregation checking is used to generate a set of group-by key-aggregation pairs. Besides, the candidate SQL queries may contain any possible selection filters. A selection filter is illustrated as a fuzzy bounding box that can be cross-validated over a group of multi-dimensional matrices, which corresponds to a conjunction of selection predicates. The full implementation is depicted in REGAL [9].

5 Group Selection

Ideally, any constructed SPJA query $\mathcal{Q}'$ should reproduce the given output table *Out*, or at least $Out \subset \mathcal{Q}'(\mathcal{D})$. Since the *Out* itself may have been skimmed by source query $\mathcal{Q}$ for a summarized version, it contains only some groups whose corresponding aggregate values are passed a threshold. However, this threshold is considered as an additional SQL functionality, and it is less being discussed in the query reverse engineering. The SQL `HAVING` clause is a specific term can be used to decide whether a set of groups will be outputted in *Out* based on the current query result $\mathcal{Q}'(\mathcal{D})$. The `HAVING` clause contains a condition which involves one or two output columns. For all groups within current query result $\mathcal{Q}'(\mathcal{D})$, a satisfied `HAVING` condition will separate them into two distinct subsets, where one subset is similar as those groups in *Out* and another subset is taken as $\mathcal{Q}'(\mathcal{D}) - Out$. On the one hand, if the `HAVING` condition involves only one column, the current query result $\mathcal{Q}'(\mathcal{D})$ is examined by all its groups are arranged based on one of the numeric columns. On the other hand, if the `HAVING` condition takes two columns, where these columns are being compared so that the *Out* exists a specified relationship between them, such as one column whose values are always larger than those from another column. A candidate query $\mathcal{Q}'$ for the motivating example in Fig. 1 is given as:

select *N.name, L.linestatus, max(O.totalprice), sum(L.quantity)*
from *Nation N, Customer C, Orders O, Lineitem L*
where *N.nationkey=C.nationkey and C.custkey=O.custkey and O.orderkey=L.orderkey and L.linenumber* > 1
group by *N.name, L.linestatus*

The generated table $\mathcal{Q}'(\mathcal{D})$ contains 50 tuples (groups). However, *Out* contains only 12 tuples (groups), and it is a subset of $\mathcal{Q}'(\mathcal{D})$. By searching through the aggregation candidates, e.g., *max(O.totalprice)* and *sum(L.quantity)*, the twelve tuples can be discerned by formulating a `HAVING` condition, as *having max(O.totalprice)* $>$ *500000*.

6 Experimental Evaluation

Implementation and Dataset. We implemented our proposed algorithm in Java with MySQL server as DBMS. The experiments were conducted on an Ubuntu machine with 2.40 GHz Intel CPU and 16 GB RAM. TPC-H benchmark is the dataset that used for experiments, with a scale factor of 1 and size of 1 GB.

TPC-H Test Queries. There are a total of 22 test queries for TPC-H benchmark. Most of them include different number of joins, except for TQ1 and TQ6, with the absence of join. We neglect the complex join query discovery, i.e. the nested joins, fk/fk joins, and equijoins, which are exhibited in TQ5 (e.g., `S_nationkey` = `C_nationkey`) and TQ21 (e.g., `L1_orderkey` = `L2_orderkey`) respectively. We test for the remaining join queries and scale them based on the number of joins, i.e. from 1 to 5.

Query Output Table Generation. Given that a test query $\mathcal{Q}$, we execute it over TPC-H dataset $\mathcal{D}$ to generate the query output table $\mathcal{Q}(\mathcal{D}) = Out$ which later the Out will be used as the input of our proposed algorithm to discover for such a query $\mathcal{Q}'$ where $\mathcal{Q}'(\mathcal{D}) = Out$. As we have selected those TPC-H benchmark queries, however, except for the join relations and join predicates are remained, other SQL operations are altered. We set several parameters for the experiments to control the variety of query output tables. For example, we will produce the query output table Out with the cardinality of m and the arity of n, and the test query $\mathcal{Q}$ contains a $\mathcal{N}$-dimensional filter.

6.1 TPC-H Join Queries

For each of these test queries, we generate a query output table with moderate row size m and column size $n = 4$ where it must contain both group-by

Table 1. Effect of number of joins.

# Joins	TQ	Tables	Runtime (s) min-max	# Graphs min-max
1	4, 12	`L, O`	144.189–281.529	1
	13, 22	`C, O`		
	14, 17, 19	`L, P`		
	15	`L, S`		
2	11	`PS, S, N`	60.937–294.624	1
	16	`P, PS, S`		
	3, 18	`C, O, L`		
3	10	`N, C, O, L`	313.882	1
4	2	`R, N, S, PS, P`	68.648–338.248	1–2
	20	`N, S, L, PS, P`		
5	7	`N1, S, L, O, C, N2`	417.508–443.197	2
	9	`N, S, L, O, PS, P`		

statements and aggregations which the number of group-by columns is set at most two while the other columns are used for aggregations. Each query contains one-dimensional filter as $\mathcal{N} = 1$. Table 1 records the experimental results based on the number of joins of TPC-H benchmark test queries. First, the inferred join table size is essentially crucial as it will impact the time cost for a table scan. Some of these test queries like TQ2, TQ11, and TQ20 contain the least inferred join table size (0.8 million tuples), which take less than 70 s for discovering these queries. Second, the number of join graphs is directly proportional to the total execution time. For example, test queries like TQ7, TQ9 and TQ20, they need to evaluate two join candidates to generate the discovered queries.

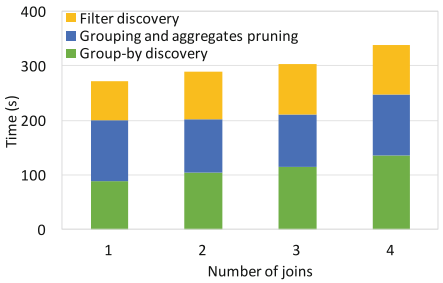


Fig. 2. Average time for query discovery against number of joins.

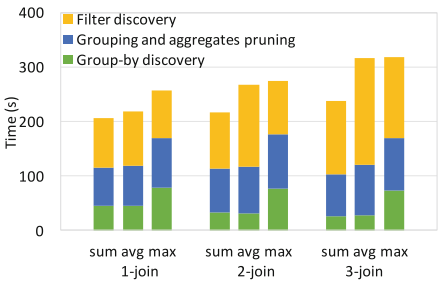


Fig. 3. Individual phase performances for different aggregations.

6.2 Individual Phase Performances vs. Joins

In order to further analyze the important factors that influence the total execution time besides the join operations, those TPC-H benchmark queries under similar considerations (i.e. the same inferred join table size) are examined. As there are multiple test queries for each number of joins, we run those queries individually and take their average running time. To avoid the cost of exploring all candidate queries, the execution time is taken once the least complex $\mathcal{Q}'$ is returned for a given test query $\mathcal{Q}$. The experimental results are illustrated as shown in Fig. 2. According to the experimental results, the total execution time is proportional to the number of joins. As the number of joins is increased, it indulges more schema tables/attributes for the query discovery and takes longer time for evaluation. Furthermore, all three individual phases involve table scans. In the phase of group-by discovery, the *Out* tuples are verified at instance-level to validate the group-by nodes. Second, during the phase of grouping and aggregates pruning, the inferred join instance is partitioned by the group-by nodes to find out the possible aggregations based on the derived constraint rules. Third, the schema attributes are used to construct the $\mathcal{N}$-dimensional matrices, so that the selection filter(s) can be found within these matrices.

6.3 Joins vs. Aggregations

Figure 3 shows the experimental results by comparing the selected aggregations for test queries w.r.t. different number of joins. Among five basic aggregate operators that we have discussed, three of them are chosen for this experiment, namely `MAX`, `SUM` and `AVG`, since `MIN` and `MAX` are symmetrical whereas `COUNT` is assumed as another `SUM` operation of a special attribute whose each of its values is set to 1. For the experiment settings, we set the parameters to output every Out with $n = 4$, where there must be one aggregation column that is selected between `MAX`, `SUM` and `AVG` with three group-by columns. First, by looking at each individual aggregation w.r.t. joins, as the number of joins is increased, the total running time is also increased. By comparing these aggregations, it is apparent that `MAX` takes the largest running time in the phase of group-by discovery regardless the number of joins if compared to both `SUM` and `AVG` aggregations. The size of group-by lattice for `MAX` is $2^4 = 16$ nodes whereas the size of group-by lattice for `SUM` or `AVG` is smaller, which is $2^3 = 8$ nodes. However, `AVG` takes more time in the phase of filter discovery as compared to `SUM` and `MAX`, since the computation for `AVG` is more complex than that `SUM`. Thus, `AVG` takes the second largest time for the query discovery.

7 Conclusion

In this paper, we bring these two main features together by integrating the promising approaches from both existing works with optimizations. Our empirical study has shown that our proposed solution can work in practice with the TPC-H benchmark dataset.

References

1. Agrawal, S., Chaudhuri, S., Das, G.: DBXplorer: a system for keyword-based search over relational databases. In: ICDE, pp. 5–16 (2002)
2. Hristidis, V., Papakonstantinou, Y.: DISCOVER: keyword search in relational databases. In: VLDB, pp. 670–681 (2002)
3. Markowetz, A., Yang, Y., Papadias, D.: Keyword search on relational data streams. In: SIGMOD, pp. 605–616 (2007)
4. Panev, K., Michel, S.: Reverse engineering top-k database queries with PALEO. In: EDBT, pp. 113–124 (2016)
5. Psallidas, F., Ding, B., Chakrabarti, K., Chaudhuri, S.: S4: top-k spreadsheet-style search for query discovery. In: SIGMOD, pp. 2001–2016 (2015)
6. Qian, L., Cafarella, M.J., Jagadish, H.V.: Sample-driven schema mapping. In: SIGMOD, pp. 73–84 (2012)
7. Qin, L., Yu, J.X., Chang, L.: Keyword search in databases: the power of RDBMS. In: SIGMOD, pp. 681–694 (2009)
8. Shen, Y., Chakrabarti, K., Chaudhuri, S., Ding, B., Novik, L.: Discovering queries based on example tuples. In: SIGMOD, pp. 493–504 (2014)
9. Tan, W.C., Zhang, M., Elmeleegy, H., Srivastava, D.: Reverse engineering aggregation queries. PVLDB **10**(11), 1394–1405 (2017)

10. Tran, Q.T., Chan, C.-Y., Parthasarathy, S.: Query by output. In: SIGMOD, pp. 535–548 (2009)
11. Tran, Q.T., Chan, C.Y., Parthasarathy, S.: Query reverse engineering. VLDB J. **23**(5), 721–746 (2014)
12. Valduriez, P.: Join indices. ACM Trans. Database Syst. **12**(2), 218–246 (1987)
13. Zhang, M., Elmeleegy, H., Procopiuc, C.M., Srivastava, D.: Reverse engineering complex join queries. In: SIGMOD, pp. 809–820 (2013)