REGAL$^+$: Reverse Engineering SPJA Queries

Wei Chit Tan[§] Meihui Zhang[¶] Hazem Elmeleegy[†] Divesh Srivastava[‡]

[§]Singapore University of Technology and Design, [¶]Beijing Institute of Technology,
[†]Amobee, [‡]AT&T Labs-Research

weichit_tan@mymail.sutd.edu.sg, meihui_zhang@bit.edu.cn,
hazem.elmeleegy@amobee.com, divesh@research.att.com

ABSTRACT

The goal of query reverse engineering is to re-generate the SQL query that produced a given result from some known database. The problem has many real world applications where users need to better understand the lineage and trustworthiness of various data reports even when the authors of those reports are no longer reachable or are unable to provide the required explanations anymore. It gets more challenging as the complexities of both the query and database schema increase. Prior work has addressed the reverse engineering of constrained types of SQL queries and sometimes on constrained schemas, such as single-table schemas. In this demonstration, we present a framework called REGAL$^+$, which builds upon, and extends prior work to enable the discovery of Select-Project-Join-Aggregation (SPJA) queries over arbitrary schemas. Without any prior schema knowledge or SQL expertise, the user only needs to upload a data report (e.g., as a spreadsheet), and the system will automatically compute and display the queries capable of generating that report from the database.

PVLDB Reference Format:
Wei Chit Tan, Meihui Zhang, Hazem Elmeleegy, and Divesh Srivastava. REGAL$^+$: Reverse Engineering SPJA Queries. *PVLDB*, 11 (12): 1982-1985, 2018.
DOI: https://doi.org/10.14778/3229863.3236240

1. INTRODUCTION

To understand structured data, numerous techniques are available via either commercial products or recent research works. However, most of them expect some prerequisites from the users, principally the skills to read the schema or to write the SQL queries. On the other hand, query reverse engineering opens up a new perspective about relational databases, where it can provide an explanation for a given table in the form of the query that was used to generate it in the first place. For example, Alice who works as a business analyst, has obtained plenty of query results that are presented in spreadsheet tables from her colleagues where some of this information is incomplete or inaccurate because either i) the original queries are not available, or ii) the created table/column names are not annotated properly. Instead of asking her colleagues individually for clarifications, since Alice is authorized to access the company database, she can re-generate those actual SQL queries from the given query results without any other specialized skills by utilizing a tool that can resolve her problem.

Since the users often construct SPJA queries that involve both joins and OLAP-style aggregations to extract insightful statistical summaries over the data warehouses, where these queries cover a wide range of SQL expressions (with multiple joins, group-by columns, aggregates and selections), it would be beneficial to them if they had access to a framework that can discover these potential queries without possessing specific knowledge about the data. Besides, there are other good side effects that result from query reverse engineering. For instance, given a query answer, users can learn about multiple equivalent SQL queries that generate the same answer, and hence improve their understanding of both SQL and the database schema.

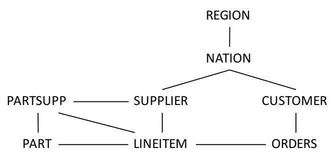
Recent works that are pertinent to reverse engineering SPJ queries over relational databases mostly depend on the schema graph, where both [7] and [5] only consider the discovery of simple join queries while [9] enables the discovery of arbitrarily complex join query graphs. Comparatively, only a few solutions such as [3, 8] focus on reverse engineering OLAP-style aggregation queries. However, these discovered queries are limited to contain a pair of columns where one column lists the distinct values for a schema attribute as groups and another column summarizes the numeric data for another schema attribute based on these groups.

Our recent work, named REGAL [6], enables the discovery of more general OLAP queries which contain multiple group-by and aggregation operations. Nevertheless, the source database is limited to a base (denormalized) table. To overcome these limitations, we extend the three-phase REGAL algorithm with join discovery to allow for reverse engineering SPJA queries, with the SQL `HAVING` clause which constrains the groups that will be present in an OLAP query output table.

In this demonstration, we showcase REGAL$^+$, a novel framework for reverse engineering SPJA queries, which adds the important join functionality to the REGAL algorithm [6]. We list our main contributions as follows:

1. We reverse engineer SPJA queries by combining the functionality of reverse engineering SPJ queries and the REGAL algorithm.

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236240



(a) Schema graph of TPC-H

```
SELECT     R_NAME, O_ORDERSTATUS, MAX(C_ACCTBAL), SUM(O_TOTALPRICE), COUNT(*)
FROM       REGION, NATION, CUSTOMER, ORDERS
WHERE      R_REGIONKEY = N_REGIONKEY AND N_NATIONKEY = C_NATIONKEY AND
           C_CUSTKEY = O_CUSTKEY AND C_ACCTBAL > 0 AND C_ACCTBAL <= 9000
GROUP BY       R_NAME, O_ORDERSTATUS
HAVING         COUNT(*) < 119000;
```

(b) SQL query

AFRICA	F	8999.87	17818116338.21	118393
AFRICA	O	8999.87	17868819851.84	118898
AFRICA	P	8999.87	1147519283.61	6238
AMERICA	F	8998.09	17951296302.56	118907
AMERICA	O	8998.09	17888552356.66	118995
AMERICA	P	8997.42	1155448262.19	6281
ASIA	P	8999.00	1181720045.64	6360
EUROPE	P	8999.10	1152682690.58	6278
MIDDLE EAST	F	8999.60	17722542616.25	118162
MIDDLE EAST	P	8993.25	1147581412.03	6215

(c) Query output table

Figure 1: An illustrative example with database schema, query and spreadsheet table.

2. We guarantee that the algorithm generates queries of the lowest complexity, i.e. it can find the smallest connected subgraph from the schema graph and the filter predicates with the lowest dimensionality.
3. We provide a visual interface to enable the users to load the query output table as an input to the system and also to explore the generated SPJA queries by interrogating the discovered groups.

2. PRELIMINARIES

Given a database $\mathcal{D}$ that consists of n relations, where each relation is denoted as $\mathcal{R}_i$, $i \in [1, n]$, a schema graph $\mathcal{SG} = (\mathcal{R}, \zeta)$ is constructed where the nodes represent the relations and the (undirected) edges represent the foreign key references. In our problem setting, a query output table *Out* is generated from $\mathcal{D}$ by an unknown SPJA query $\mathcal{Q}$. The first goal is to identify a minimal join subgraph $\mathcal{J}(\mathcal{G})$ of the schema graph $\mathcal{SG}$, such that it contains multiple schema tables where its leaves should cover every column of *Out*. The set of schema attributes of the relations in this minimal join subgraph is denoted as $\mathcal{A}$. The subsequent goal is to discover the group-by and aggregations with the optional selection and `HAVING` conditions that are applied on $\mathcal{J}(\mathcal{G})$. Specifically, we express SPJA queries in SQL as:

`SELECT`	A_1 `AS` Out_1 , ... , A_k `AS` Out_k,
	$\mathcal{F}_1$ `AS` Out_{k+1} , ... , $\mathcal{F}_m$ `AS` Out_{k+m}
`FROM`	$\mathcal{J}(\mathcal{G})$
`WHERE`	$\mathcal{J}_1 \wedge ... \wedge \mathcal{J}_\gamma \wedge \mathcal{C}_1 \wedge ... \wedge \mathcal{C}_\ell$
`GROUP BY`	$A_1, ..., A_k$
`HAVING`	$\mathcal{F}_i$ *op* ρ

The `SELECT` clause is used to determine each column of *Out*, as $Out_i, i \in [1, k+m]$ in which $A_i, i \in [1, k]$ is a group-by attribute that corresponds to $Out_i, i \in [1, k]$ and $\mathcal{F}_i, i \in [1, m]$ is an aggregation that corresponds to $Out_i, i \in [k+1, k+m]$. Each aggregation $\mathcal{F}_i$, is equal to $AGG(A)$ where AGG is one of the basic aggregate operators, i.e. `COUNT`, `SUM`, `AVG`, `MAX`, and `MIN`, which can be applied on any relational attribute, $A \in \mathcal{A}$. The `WHERE` clause comprises the conjunction of both join predicates and selection predicates. The join predicates are denoted as $\mathcal{J}_1 \wedge ... \wedge \mathcal{J}_\gamma$ where each $\mathcal{J}_i, i \in [1, \gamma]$ is one of the edges from $\mathcal{J}(\mathcal{G})$ and γ is the number of the edges in the subgraph $\mathcal{J}(\mathcal{G})$. The selection predicates are denoted as $\mathcal{C}_1 \wedge ... \wedge \mathcal{C}_\ell$ where each $\mathcal{C}_i, i \in [1, \ell]$ is expressed as A *op* X, where A is a relational attribute, *op* is a comparison operator and X is a constant value. As in the REGAL algorithm [6], filter discovery is based on numerical attributes for a conjunction of range predicates. The `HAVING` clause contains a condition $\mathcal{F}_i$ *op* ρ, where $\mathcal{F}_i, i \in [1, m]$ is one of the aggregation columns that is controlled by a parameter ρ, which can be either a constant value X or another aggregation column $\mathcal{F}_j, j \neq i$ and $j \in [1, m]$. Figure 1 gives an illustrative example of a query output table *Out* that is generated from the TPC-H database $\mathcal{D}$ with its schema graph $\mathcal{SG}$ and its generating query $\mathcal{Q}$ where $\mathcal{Q}(\mathcal{D}) = Out$.

3. SYSTEM ARCHITECTURE

Figure 2 shows the architecture of our system. For a given database, the column-level inverted indexes are precomputed over all relational attributes to optimize the query search. The system takes a spreadsheet table as input which is provided by the user. The input spreadsheet, denoted as query output table *Out*, is analyzed by the system given a database instance $\mathcal{D}$ whose schema graph $\mathcal{SG}$ is used to discover the join queries. The system operates in a step-by-step fashion, where the schema of *Out* has to be first outlined through the column mapping. After that, given the schema graph, a set of minimally connected subgraphs is generated through join discovery. For the remaining query discovery, the prior work of REGAL [6] is applied to find out both OLAP group-by and aggregation candidates, together with the possible filter predicates. Nevertheless, if the discovered queries generate additional groups which are not present in *Out*, they are filtered out by adding `HAVING` conditions to the discovered queries upon any constrained output columns.

During the column mapping, every column $Out_i, i \in [1, k+m]$ is examined to find a corresponding set of schema attributes where each attribute is denoted as $\phi(Out_i) = A$, and it covers all distinct column values. For each $\phi(Out_i)$, its source relation thus becomes the node that must be included in the candidate subgraphs. If there is a duplicate $\phi(Out_i)$, it means that a pair of similar nodes must be present in the candidate subgraphs so that the schema table/attribute is joined twice to generate the query output table *Out*.

3.1 Distinct Tree Semantics

Determining the join tables and the join predicates are the most crucial tasks in join discovery. Among the various possibilities, the schema-based approach is known as a very promising solution which takes the database schema as a graph and does not perform any join execution for candidate enumeration. In this paper, we adopt the distinct tree semantics (as defined in [4]), which was first proposed by DISCOVER keyword search [1], and later implemented in most of the recent works for reverse engineering queries (see [2, 5, 7, 8]).

In this approach, given a schema graph $\mathcal{SG}$, we take a table $\mathcal{R}_i$ as a leaf node since $\phi(Out_i) = \mathcal{R}_i.A$ so that there will be a set of leaf nodes to be used to generate the candidate subgraphs. To guarantee the candidate subgraphs are minimally connected, one of the leaf nodes is selected as the root node to explore the schema graph via breadth first search traversal in order to connect the other leaf nodes to form a candidate subgraph $\mathcal{J}(\mathcal{G})$. If there are duplicate leaf nodes, the graph nodes are allowed to be visited more than

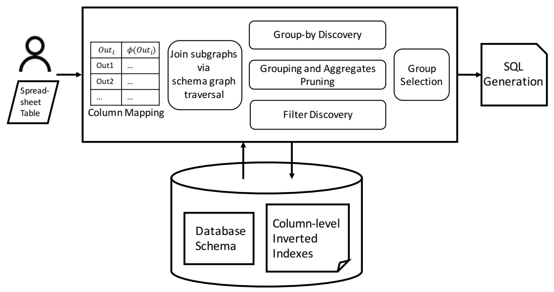


Figure 2: System architecture.

once to compute the candidate subgraphs. Note that we do not determine the arbitrary join graphs using the distinct root semantics [9], where each leaf node is used as a root to build a tree up to the pre-defined depth and a candidate graph is built upon a starred node among the trees that interconnects all disjoint paths from the leaf nodes.

Moreover, one needs to consider the possibility that one or more columns of Out is/are not related with any schema attribute for the column mapping. Thus, the current schema of a candidate subgraph is explored to determine whether these columns can be described as aggregations, especially for the aggregates that transform a set of values into a new scalar value, such as `AVG`, `SUM` and `COUNT`. For any schema attribute A, its statistical properties are examined for the possibility that $\mathcal{F}_i = AGG(A)$, where AGG is an aggregate operator. The current schema must satisfy all these prerequisites to generate the given Out; otherwise an alternate option is to expand this candidate subgraph into its individual neighbour nodes until the desired schema is found.

3.2 Instance Verification via Group-by Lattice

The resulting schema is used to compute a join instance using the join indices. It is important to verify this join schema by checking the rows of Out against the join instance, which can be very costly if it requires table scans. Since it involves OLAP-style aggregations, constructing a group-by lattice as proposed by REGAL [6] helps to reduce this cost by quickly pruning the invalid group-by candidates. A group-by lattice is built upon a set of schema attributes from the column mapping, where its nodes represent all possible combinations of group-by columns and its edges represent the parent-child relationships between two nodes. A parent node indicates a combination of group-by columns that can subsume its children nodes.

Before the instance verification, some invalid candidate nodes can be pruned by keyness checking, which is used to check the uniqueness of group-by columns of Out at the instance level. For a node that passes the keyness check, containment checking is used to prove that its group-by tuples can be output from the join instance, which is also the instance verification. The containment check makes its traversal from the lattice root in a top-down fashion, where if an ancestor node passes the check, all its descendants will also pass without extra checks. Besides, in order to minimize the number of table scans, the group-by tuples of a candidate node are clustered into minimal sets where each set of group-by tuples contains a single column tuple instance. Thus, among those group-by columns, the specific column with the fewest distinct tuples is used to cluster the others at row-level. By knowing the row indexes of a single column tuple instance, these indexes are used to find out the clustered group-by tuples; however, the group-by node is considered invalid if any one of these tuples isn't found. The invalid tuples thus become the negative witness tuples to speed up the candidate assessment.

3.3 Group Selection

Once the group-by candidates are confirmed, the query discovery process continues for OLAP aggregations and filter predicates. However, the discovered queries might not be enough to compute Out as their results may represent a superset of Out. In this case, it is necessary to introduce a new SQL clause in the query discovery, namely the `HAVING` clause which is used to specify a selection condition for the groups of Out, using $\mathcal{F}_i$ op ρ, where ρ is a parameter that controls an aggregate column $\mathcal{F}_i$, so that any group whose $\mathcal{F}_i$ value satisfies the condition will be output in Out.

To guarantee the correctness of a candidate SQL query, all the tuples of Out must be included in its current query answer, and this current query answer must be examined to remove any extra groups that are not covered by Out through the discovery of `HAVING` condition. Intuitively, by using the current query answer that will output a superset of Out, denoted as $\mathcal{Q}'(D)$, the groups of $\mathcal{Q}'(D)$ are sorted by each individual aggregate column respectively. Hence, if the `HAVING` condition can be determined in the case that the subset ($Out \subset Q'(D)$) is found either within or beyond a certain threshold, then it can be determined as `HAVING` $\mathcal{F}_i$ op X, where op is a comparison operator and X is a constant.

Alternatively, the parameter ρ can also be another aggregate column, $\mathcal{F}_j$. In lieu of sorting the groups based on an individual column, two aggregate columns are compared pairwise, so that there will be a clear relationship to distinguish the subset ($Out \subset Q'(D)$). Consequently, this condition can be stated as `HAVING` $\mathcal{F}_i$ op $\mathcal{F}_j$.

4. SYSTEM DEMONSTRATION

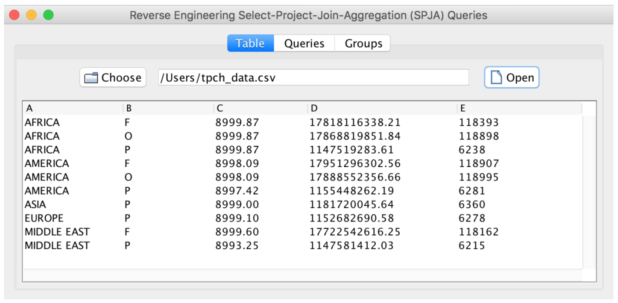
Our REGAL$^+$ framework is implemented in Java with Swing using WindowBuilder to create an interface that runs on top of MySQL. In this demonstration, we showcase how REGAL$^+$ can efficiently regenerate SPJA queries from the TPC-H benchmark dataset, with the scale factor of 1 (1GB size). We design the user interface by separating the functionalities into three aspects: first, the spreadsheet table that is loaded and can be viewed; second, the whole process of query discovery will be shown; third, the OLAP groups in the spreadsheet table can be analyzed.

4.1 Loading Spreadsheet Table

First, to initiate the REGAL$^+$ framework, the user loads a spreadsheet table that might be exported from any previous SQL result in other formats. After loading the spreadsheet table, it can be viewed again in the user interface for reference, as shown in Figure 3. For demonstration purpose, we use the spreadsheet example in Figure 1(c) to showcase how our proposed framework can discover SQL queries that are similar to the query example in Figure 1(b). Note that the column names are replaced by the characters in alphabetical order to avoid any misinterpretation, assuming their original names might be mislabeled.

4.2 Query Discovery

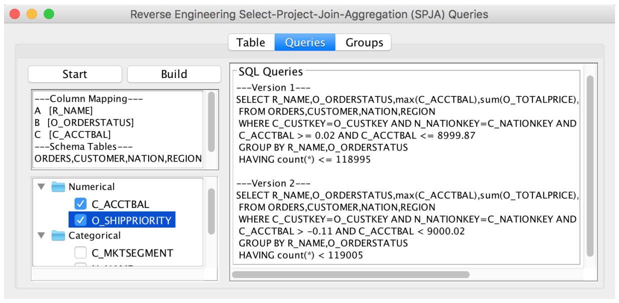
In the second tab "Queries", the process of query discovery begins by clicking the button "Start". In order to give


Reverse Engineering Select-Project-Join-Aggregation (SPJA) Queries
Table
Queries
Groups
Choose
Open

/Users/tpch_data.csv

A	B	C	D	E
AFRICA	F	8999.87	17818116338.21	118393
AFRICA	O	8999.87	17868819851.84	118898
AFRICA	P	8999.87	1147519283.61	6238
AMERICA	F	8998.09	17951296302.56	118907
AMERICA	O	8998.09	17888552356.66	118995
AMERICA	P	8997.42	1155448262.19	6281
ASIA	P	8999.00	1181720045.64	6360
EUROPE	P	8999.10	1152682690.58	6278
MIDDLE EAST	F	8999.60	17722542616.25	118162
MIDDLE EAST	P	8993.25	1147581412.03	6215

Figure 3: Loading spreadsheet table.


Reverse Engineering Select-Project-Join-Aggregation (SPJA) Queries
Table
Queries
Groups
Start
Build
SQL Queries

```
---Column Mapping---
A  [R_NAME]
B  [O_ORDERSTATUS]
C  [C_ACCTBAL]
---Schema Tables---
ORDERS,CUSTOMER,NATION,REGION
```

Numerical
- [✓] C_ACCTBAL
- [✓] O_SHIPPRIORITY

Categorical
- [ ] C_MKTSEGMENT

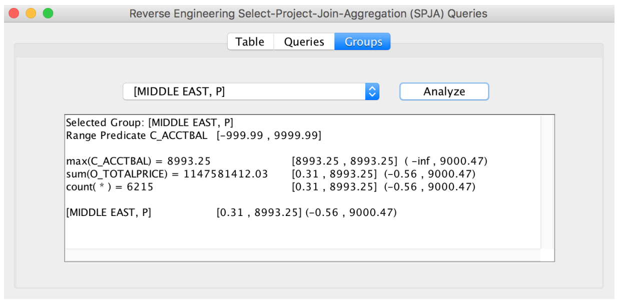
```
---Version 1---
SELECT R_NAME,O_ORDERSTATUS,max(C_ACCTBAL),sum(O_TOTALPRICE),
 FROM ORDERS,CUSTOMER,NATION,REGION
 WHERE C_CUSTKEY=O_CUSTKEY AND N_NATIONKEY=C_NATIONKEY AND
 C_ACCTBAL >= 0.02 AND C_ACCTBAL <= 8999.87
 GROUP BY R_NAME,O_ORDERSTATUS
 HAVING count(*) <= 118995

---Version 2---
SELECT R_NAME,O_ORDERSTATUS,max(C_ACCTBAL),sum(O_TOTALPRICE),
 FROM ORDERS,CUSTOMER,NATION,REGION
 WHERE C_CUSTKEY=O_CUSTKEY AND N_NATIONKEY=C_NATIONKEY AND
 C_ACCTBAL > -0.11 AND C_ACCTBAL < 9000.02
 GROUP BY R_NAME,O_ORDERSTATUS
 HAVING count(*) < 119005
```

Figure 4: Query discovery.

the user some preliminary details, the column mapping and the discovered join schema are delineated in the top-left section. Hence, she can gain some basic understanding about the database schema. Subsequently, the bottom-left section lists all the schema attributes for filter discovery, which are divided into either categorical or numerical attributes. The user can optionally select and narrow down the intended schema attributes to generate the filter predicates. After the selection, the button "Build" is clicked to build the SQL queries. As the purpose is to discover the SPJA queries, the discovered queries support multiple SQL features like joins, selections, aggregates (e.g., `MAX`, `MIN`, `AVG`, `SUM` and `COUNT`), the `GROUP BY` clause and the `HAVING` clause.

For the discovered queries that contain range predicates, we show them in two different versions, which are shown in the right section. The first version represents the tightest queries and contains the range predicates with comparison operators like "$\geq$" and "$\leq$", which means that the inner fuzzy bounding box (see [6] for more details) of a multi-dimensional filter is used in the `WHERE` clause. In contrast, the second version represents the most loose queries and contains the range predicates with comparison operators like "$>$" and "$<$", which means that the outer fuzzy bounding box of a multi-dimensional filter is used in the `WHERE` clause. By learning about these two extremes, the user can come up with other valid, and perhaps more meaningful, SQL queries to re-generate the data in the input spreadsheet.

Figure 4 illustrates the above descriptions. From the discovered SQL queries, both versions are valid if compared to the query example in Figure 1(b) since they all have the same query structure, i.e., the join structure and aggregations. Besides, the discovered SQL queries can be validated through the constant values used for the range predicates. For instance, the constant values of the predicate "$C\_ACCTBAL$" are placed within both inner and outer bounding boxes, for example, the value $0 \in (-0.11, 0.02]$ and the value $9000 \in [8999.87, 9000.02)$. Therefore, the user can


Reverse Engineering Select-Project-Join-Aggregation (SPJA) Queries
Table
Queries
Groups
[MIDDLE EAST, P]
Analyze

```
Selected Group: [MIDDLE EAST, P]
Range Predicate C_ACCTBAL  [-999.99 , 9999.99]

max(C_ACCTBAL) = 8993.25                 [8993.25 , 8993.25]  ( -inf , 9000.47)
sum(O_TOTALPRICE) = 1147581412.03        [0.31 , 8993.25]  (-0.56 , 9000.47)
count( * ) = 6215                        [0.31 , 8993.25]  (-0.56 , 9000.47)

[MIDDLE EAST, P]            [0.31 , 8993.25] (-0.56 , 9000.47)
```

Figure 5: Group interpretation.

opt for either of these discovered queries for further analysis. Since this step involves the overall query discovery process, it took over a minute to discover the queries in Figure 4. The system performance depends on not only the size of database instance, but also the query structures being used in the input spreadsheet. An extensive study over a single relation from TPC-H dataset can be found in [6].

4.3 Group Interpretation

In the third tab "Groups", the user can select an OLAP group presented in the spreadsheet table and then click the button "Analyze" to inspect each aggregate result for the selected group.

Every aggregate value within the selected group is analysed for more details. This is because in the process of filter generation, each aggregate value could have its own inner and outer fuzzy bounding boxes, and all of these fuzzy bounding boxes are intersected to finalize the filter predicates, which are included in the discovered queries. By knowing this information, the user can verify the correctness of both data and queries. Furthermore, for each aggregate result corresponding to a given group, we report both the maximum and minimum feasible regions of the schema attributes that are used for selection conditions. Figure 5 depicts the analyzed results for a selected OLAP group.

5. CONCLUSION

In summary, our REGAL$^+$ framework demonstrates both the practicality and usefulness of providing complex query reverse engineering features to database users, which would enhance database usability in general.

6. REFERENCES

[1] V. Hristidis and Y. Papakonstantinou. DISCOVER: keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[2] D. V. Kalashnikov, L. V. S. Lakshmanan, and D. Srivastava. Fastqre: Fast query reverse engineering. In *SIGMOD*, pages 337–350, 2018.

[3] K. Panev and S. Michel. Reverse engineering top-k database queries with PALEO. In *EDBT*, pages 113–124, 2016.

[4] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: the power of RDBMS. In *SIGMOD*, pages 681–694, 2009.

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

[6] W. C. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava. Reverse engineering aggregation queries. *PVLDB*, 10(11):1394–1405, 2017.

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

[8] Q. T. Tran, C. Y. Chan, and S. Parthasarathy. Query reverse engineering. *VLDB J.*, 23(5):721–746, 2014.

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