Utah State University

DigitalCommons@USU

All Graduate Theses and Dissertations | Graduate Studies

8-2019

Query AutoAwesome

Chetna Suryavanshi
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/etd

 Part of the Computer Sciences Commons

Recommended Citation

Suryavanshi, Chetna, "Query AutoAwesome" (2019). *All Graduate Theses and Dissertations*. 7546.
https://digitalcommons.usu.edu/etd/7546

This Thesis is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.

Query AutoAwesome

by

Chetna Suryavanshi

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Curtis Dyreson, Ph.D. Major Professor	Stephen Clyde, Ph.D. Committee Member
Haitao Wang, Ph.D. Committee Member	Richard S. Inouye, Ph.D. Vice Provost for Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2019

Copyright © Chetna Suryavanshi 2019

All Rights Reserved

ABSTRACT

Query AutoAwesome

by

Chetna Suryavanshi, Master of Science

Utah State University, 2019

Major Professor: Curtis Dyreson, Ph.D.
Department: Computer Science

Query AutoAwesome is an algorithm to enhance an existing SQL query. Writing a query can be a time consuming process. Once written, a query encodes how to retrieve data from a database. As the size of the data increases, a query writer may be interested in learning about improvements to a query. Query AutoAwesome provides automatic enhancement of SQL queries. Query AutoAwesome ingests a query, database schema, and the data to generate an enhanced query. The query is modeled as a query pattern with configurable parameters, a list of columns in a SELECT clause. Our algorithm modifies the pattern and the parameters to generate new queries. Each new query is scored using an objective function and the top-k queries are chosen for further enhancements. This process is repeated until satisfactory results are achieved.

(53 pages)

PUBLIC ABSTRACT

Query AutoAwesome

Chetna Suryavanshi

This research investigates how to improve *legacy queries*. Legacy queries are queries that programmers have coded and are used in applications. A database application typically has tens to hundreds of such queries. One way to improve legacy queries is to add new, interesting queries that are similar to or based on the set of queries. We propose Query AutoAwesome, a tool to generate new queries from legacy queries. The Query AutoAwesome philosophy is taken from Google's AutoAwesomizer tool for photos, which automatically improves a photo uploaded to Google by animating the photo or adding special effects. In a similar vein, Query AutoAwesome automatically enhances a query by ingesting a database and the query. Query AutoAwesome produces a set of enhanced queries that a user can then choose to use or discard. A key problem that we solve is that the space of potential enhancements is large, so we introduce objective functions to narrow the search space to a tractable space. We describe our plans for implementing Query AutoAwesome and discuss our ideas for future work.

To my family and friends....

ACKNOWLEDGMENTS

I would like to offer my sincere gratitude to my thesis advisor, Dr. Curtis Dyreson. I am grateful for his unfailing support and patience with me in answering my questions and guiding me in the right direction. It was all his guidance and assistance that helped me through the outcome of this thesis.

Besides my advisor, I would like to thank my committee members, Dr. Haitao Wang and Dr. Stephen Clyde, for being great mentors as they are.

In addition, I would like to thank my family and friends, for supporting me throughout the duration of my academic pursuits and through my life in general.

Chetna Suryavanshi

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

This research is inspired by Google's Auto Awesome app. Auto Awesome is an app that produces creations such as edited photos, collages, animated pictures and movies based on the photos and videos uploaded to Google Photos[1] using special effects or by combining photos as shown in Figure 1.1. For example, a family picture with snow can be enhanced with falling snow effect. The same scenario can be augmented using auto-awesome enhancements such as creating slide shows with music and geo-tagging. In this completely automated process, human interaction would play a role only to decide whether to save or discard the enhancements. Furthermore, only a few photos will be selected for enhancements from the collection.

The above-mentioned auto-awesome philosophy is recommended to be included in a relational database application (DBApps). DBApps allows to manage and query data in a relational database through collection of forms and processing scripts via a three-tier architecture (client, web server, and database server) as shown in Figure 1.2. A DBApp application starts with small core of database, tables and forums that develops into a project consisting of a complex number of tables and scripts. Hence, auto-awesome can help to use this information to create a crisp and necessary result in the development.

Domain experts, who maintain the DBApp, are constantly looking for amendments to modify, improve or extend some features of the software. As domain experts are limited with their programming knowledge, a dire need arises for them to have a tool that can perform the tasks for them without them having to learn programming. These expected modifications can be achieved by domain experts with the use of software that can perform automated or semi-automated modifications and extensions, code error identification, and refactoring opportunity identification.

[1] Google re-branded the app as "Photo Creations" in 2015.

Fig. 1.1: AutoAwesomized Photos: upper left-Christmas lighting effect, upper middle - postcard effect, upper right - snowflakes falling effect, lower - panorama constructed from three photos

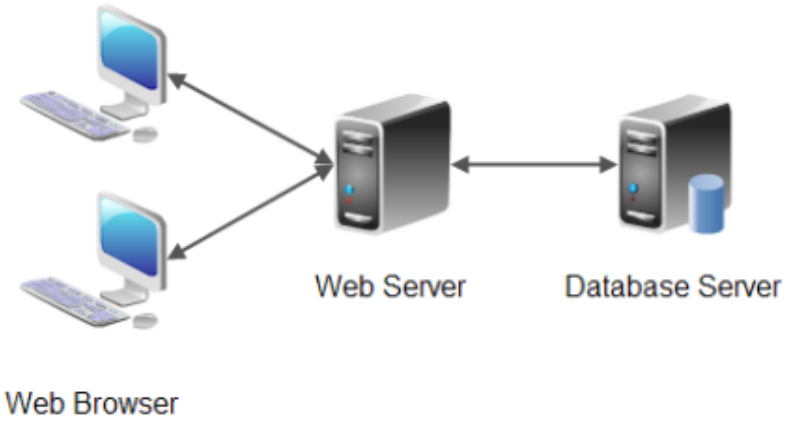

Fig. 1.2: Three-tier architecture: client, web server, and database server

A key part of a DBApp is *querying*. A DBApp typically has tens to hundreds of queries. A query is a script or program to retrieve data that meets some conditions from a database. Queries are the most frequent database operation, typically occurring far more frequently than update. Programmers invest time and effort into developing a set of queries that meet the needs of the database's end-users. As new users and new user needs arise over time, new queries must be developed.

We propose applying an Auto Awesome philosophy to the pool of queries in a system that we call Query AutoAwesome (QAA). QAA ingests a query, the database schema, and the data and produces an *enhanced* query. QAA can enhance a query in several ways.

- *Alternative literals enhancement* - Literals in a WHERE clause predicate can be enhanced by providing a user with a menu of alternatives. For instance, suppose a query in a customer database selects for customers in `Portland` using the `WHERE` clause predicate given below.

  ```
  City = 'Portland'
  ```

  The query can be enhanced by changing the literal, *e.g.,* to `'San Francisco'`, via a drop-down menu or by utilizing ranges available in related columns, *e.g.,* cities in a given country.

  Generalizing this approach, we propose a *parameterized query* where the literals in a query are specified by parameters, *e.g.,* a query Q to find customers in `'Portland'` becomes the query $Q(c)$ where c is the city parameter.

- *Table/column enhancement* - Tables and columns can also be enhanced. Suppose that a query relates an `Order` table to a `Product` table *i.e.,* the table is one created from a many-to-many relationship type between `Customer` and `Product` entity types. Other relationship types, may also relate the same pair of tables and are candidates for replacing the `Order` table. Effectively the table also becomes a parameter for the query. At the same time, columns specific to the table being replaced should also become parameters.

- *Query combination enhancement* - Some queries can be combined. For instance, a query for particular `Order` by a particular `Customer` can be combined with a query for `Shipment` or `Supplier` details to correlate `Order`s with `Shipment`s or `Supplier`s.

- *Column padding enhancement* - Queries that project only some columns can be expanded to include more columns (leading to increased query combination potential).

- *Group by padding enhancement* - Grouping and aggregation can be similarly padded. For instance, a query to count the number of `Customer`s per `Country` can be transformed to count the number of `Customer`s per the combination of `City` and `Country`.

These are a few of the enhancement provided by QAA. In general our approach will be to take a query and enhance it using some technique. An enhanced query can then be further enhanced using another technique. Since there are many possible enhancements, the space of potential enhancements is large. To improve efficiency and select only the best enhancements, we use objective functions to measure the *quality* of an enhancement. Each enhancement is measured. When a sufficient measure has been achieved the process stops and delivers the enhanced query, along with a short description of the enhancement.

This thesis is organized as follows. We first present a generalized SQL model representing a query through a parameterized model and its description. We also discuss objective functions for comparing queries. Next, we discuss how an existing query can be modified based on the presence or absence of a parameter. A query is modified by potentially transforming it using a WHERE clause transform, Aggregate transform, Add/Remove Columns transform, or HAVING clause transform. The WHERE clause transform discusses ways to modify the clause by adding or removing an expression or using a subquery. The Aggregate transform shows how to add, remove or modify an aggregate. The Add/Remove Columns transform goes into modifying a query based on the columns projected. The HAVING clause transform performs a similar modification to the WHERE clause transform. We then present related work and our plans for implementation. The paper concludes with a discussion of future work.

CHAPTER 2

Model

In this chapter we present a model for QAA for SQL. We focus on SQL because it is the most popular database language, but our approach is general enough to apply to other languages.

A simple model of an SQL query is given below. Let $Q(\overline{C}, \overline{T}, \overline{G}, \Theta_H, \Theta_W)$ be a query parameterized by the following.

- $\overline{C} = [c_1, \ldots, c_n]$ is a list of columns that are projected.
- $\overline{T} = [t_1, \ldots, t_m]$ is a list of tables that contain the data.
- $\overline{G} = [g_1, \ldots, g_k]$ is a list of grouping columns.
- Θ_H is a predicate on the groups.
- Θ_W is a selection predicate.

The parameters map to the following SQL query template.

```
SELECT c_1, ..., c_n
FROM t_1, ..., t_m
WHERE Θ_W
GROUP BY g_1, ..., g_k
HAVING Θ_H
```

Note that this query template does not include operations between tables, such as `UNION`, nor clauses in a query that modify the form of the result, such as `LIMIT` and `ORDER BY`. Since clauses can be optional, an empty parameter signifies that a clause is absent.

A query can be enhanced by modifying a parameter, such as removing or adding columns from $\overline{C}$ or changing the where clause predicate. We utilize an objective function

to rank enhancements and prune low quality enhancements: $I : Q \times Q \times D \Rightarrow R$ where Q is the set of queries, D is the set of databases, and R is the real numbers. Let Q_1 and Q_2 are enhanced versions of Q. Then Q_1 is ranked higher if $I(Q, Q_1, D) > I(Q, Q_2, D)$.

QAA utilizes objective functions as defined by the user. Below are two objective functions that we use in this paper:

- **Diversity objective function** - Maximizes the number of *new* tuples.
  - For an enhanced query, Q_e, with no schema change, the intuition is that Q_e's ranking depends on how many new tuples are produced.

$$I(Q, Q_e, D) = |\mathbf{eval}(Q_e, D) - \mathbf{eval}(Q, D)|$$

  - If Q_e changes the schema, we cannot take the difference between two relations so the objective function measures how many tuples are produced with respect to the original query.

$$I(Q, Q_e, D) = |\mathbf{eval}(Q_e, D)| - |\mathbf{eval}(Q, D)|$$

- **Consistency objective function** - Keeps the number of tuples the same. The intuition is that an enhanced query should return roughly the same number of tuples as the original query.
  - For a new query with no schema change:

$$I(Q, Q_e, D) = \frac{1}{\mathbf{abs}(|\mathbf{eval}(Q, D) - \mathbf{eval}(Q_e, D)|) + 1}$$

  - For a new query with schema change:

$$I(Q, Q_e, D) = \frac{1}{\mathbf{abs}(|\mathbf{eval}(Q, D)| - |\mathbf{eval}(Q_e, D)|) + 1}$$

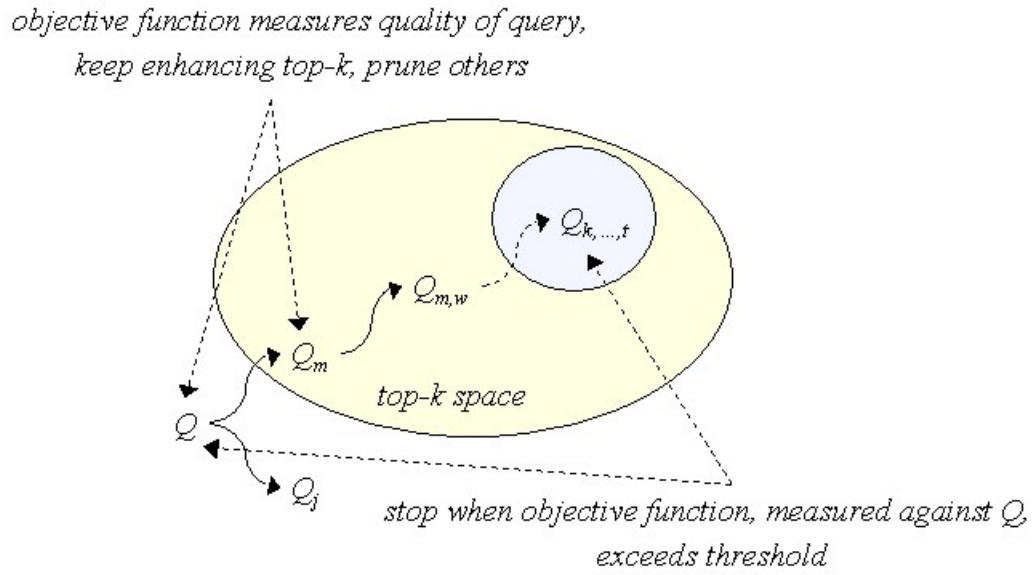

Fig. 2.1: Exploring the search space of enhanced queries

Figure 2.1 gives an overview of the strategy used by QAA to explore the search space. In each round, QAA generates a set of enhanced queries. An objective function is used to measure the enhancement with respect to the original query. The top-k enhanced queries are chosen for further enhancements in the next round. The process stops when a sufficient objective function score is achieved by some enhanced query and those queries are presented to the user as the set of enhanced queries.

CHAPTER 3

Query AutoAwesome Algorithm

In this chapter we describe our algorithm for enhancing a query. We enhance a query by ingesting the query and a database, both the schema and data. We will modify one or more parameters for the query based on the transformation technique applied. Each transformed query is scored using an objective function, and queries that score highest are kept.

As a running example we will utilize the following schema.

```
Customers(CID, Name, ContactName, Country, City, PostalCode)
Orders(OID, CID, OrderDate, EmpID, ShipperID)
OrderDetails(OrderDetailID, OrderID, ProductID, Quantity)
Products(ProductID, ProductName, Price)
Employees(EmployeeID, LastName, FirstName, BirthDate, Photo)
Shippers(ShipperID, ShipperName, Phone)
```

We will assume that the `Customers` relation has 100 tuples, while there are 400, 518, 77, 10 and 3 tuples in `Orders`, `OrderDetails`, `Products`, `Employees` and `Shippers` respectively.

3.1 WHERE Clause Transform

The WHERE clause can be transformed in several ways. This section presents the potential transformations that QAA uses. The flowchart in Figure 3.1 shows the potential transformation pathways. If the query has a subquery then the subquery can be modified or removed, or the WHERE clause (apart from the subquery) can be modified. If the query lacks a subquery it can be added or the WHERE clause expression can be modfied.

In general, the WHERE clause transform modifies the predicate in the WHERE clause.

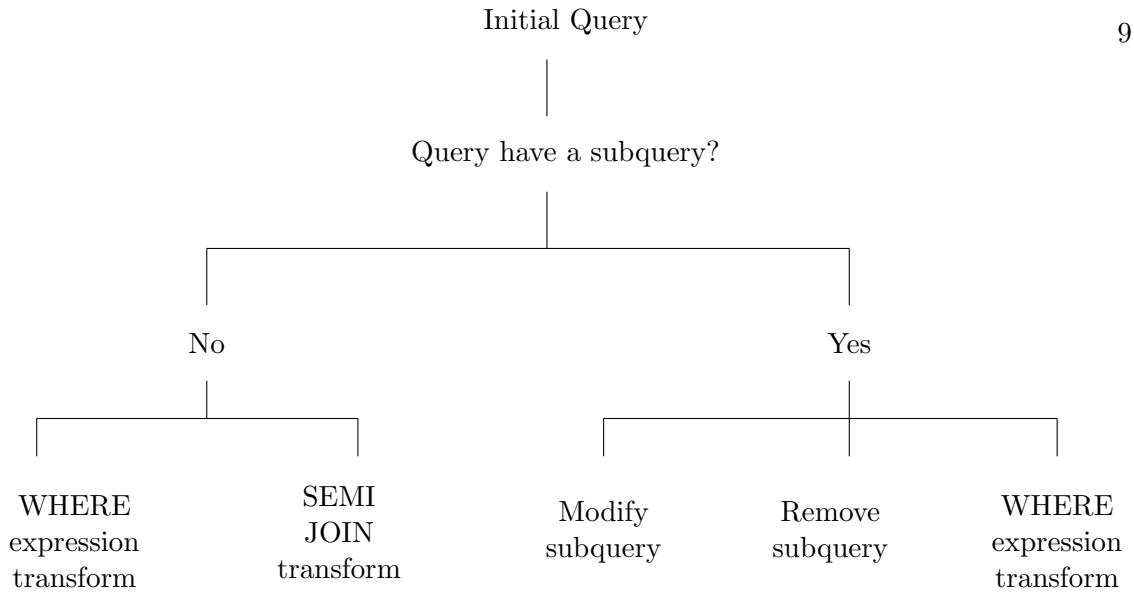

Fig. 3.1: WHERE condition Processing Tree

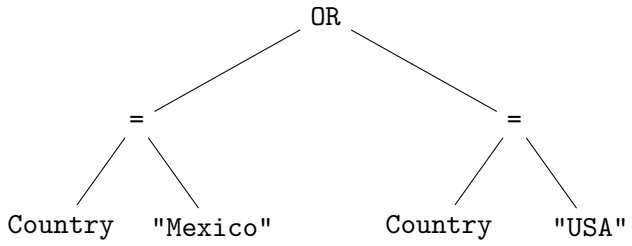

Fig. 3.2: Expression tree for `WHERE Country = "Mexico" OR Country = "USA"`

Definition 3.1.1 *The WHERE clause transform, $\mathcal{W}$, transforms the WHERE clause predicate as follows.*

$$\mathcal{W}(Q(\overline{C}, \overline{T}, \overline{G}, \Theta_H, \Theta_W), D) = Q(\overline{C}, \overline{T}', \overline{G}, \Theta_H, \Theta'_W)$$

■

Θ_W can be expressed as an expression tree, T, where each node in T is an operation and each leaf is a literal or the name of a column. For example, if the WHERE clause in a query is

```
WHERE Country = "Mexico" OR Country = "USA"
```

then Θ_W is the tree shown in Figure 3.2.

3.1.1 WHERE expression transform

The WHERE clause can be transformed by one or more applications of an *Inserter Deleter*, or *Modifier* pattern, either adding, removing, or changing an expression in the tree.

- Inserter - Let some node in Θ_W be (o, t_L, t_R) represent an expression where o is an operation, *e.g.,* `AND`, `OR` or $=$, and t_L and t_R are the left and right operands (children), respectively. The Inserter mutator modifies the node by inserting an expression to the right child as follows $(o, t_L, (o', t_R, t_N))$ where t_N is a new expression and o' is either `AND` or `OR`. The left child insertion is similar. For the WHERE clause transform we only consider inserting conjunction (`AND`), disjunction (`OR`), or equality comparison (`=`) expressions. We omit other comparisons, such as inequality and string matching to reduce complexity. As an example, suppose the WHERE clause includes the following comparison.

  ```
  WHERE Country = "Mexico" OR Country = "USA"
  ```

  Then we can insert an additional comparison into the clause as follows.

  ```
  WHERE Country = "Mexico"
    OR (Country = "USA" AND City = "Chicago")
  ```

  The expression tree changes from Figure 3.2 to Figure 3.3.

  The choice of which comparison to insert into Θ_W depends on the objective function. Suppose that we are using the Diversity objective function, then adding a conjunct does not help since the filter becomes more selective. For example, adding

  ```
  AND City = "Chicago"
  ```

  further reduces the number of existing tuples produced by Θ_W and therefore does not add diversity to the original query as shown in Venn diagram Figure 3.4. On the other hand, adding a disjunctive comparison may or may not add diversity depending on the comparison as shown in Venn diagram Figure 3.5. For example, adding

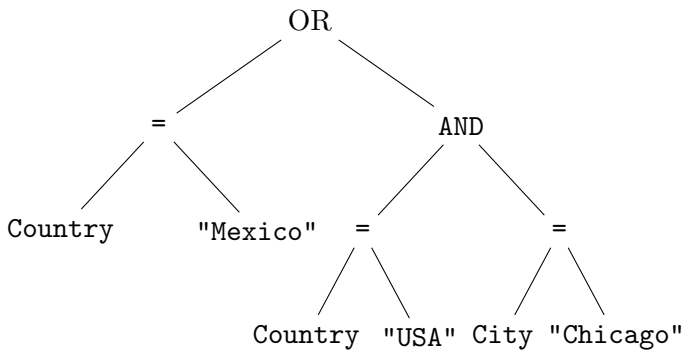

Fig. 3.3: Inserting into the WHERE clause

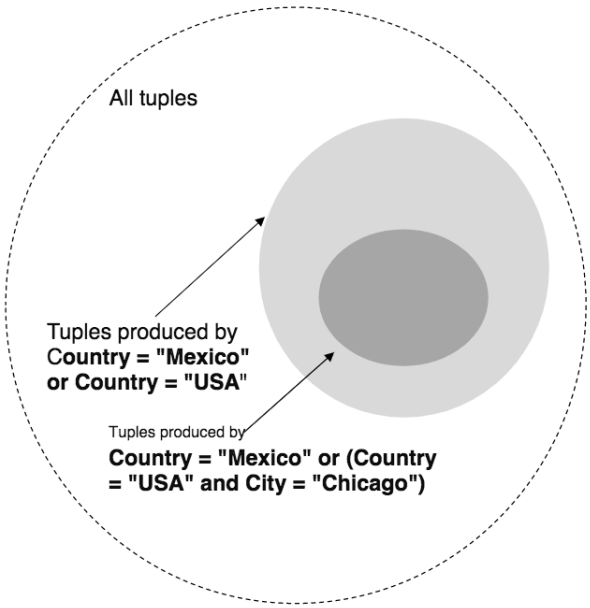

Fig. 3.4: Venn Diagram showing count of tuples on adding a condition with conjunct

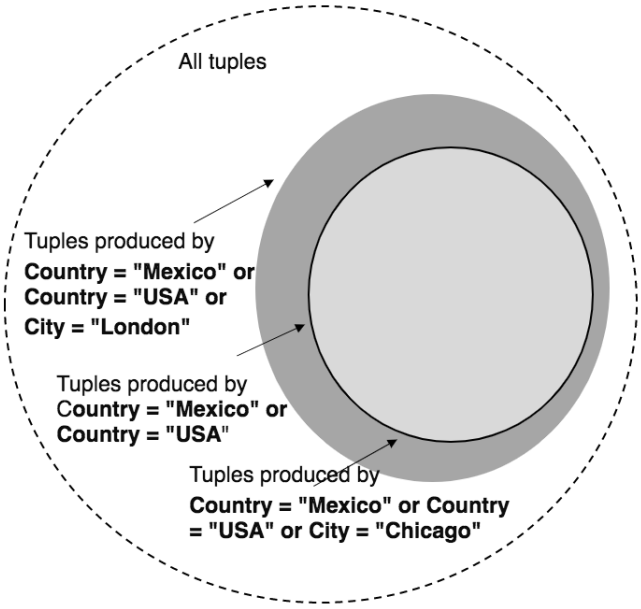

Fig. 3.5: Venn Diagram showing count of tuples on adding a condition with disjunct

```
OR City = "Chicago"
```

does not add any diversity since the query already selects tuples in the USA and Chicago is a city in the USA, while adding

```
OR City = "London"
```

would likely increase the number of tuples produced, assuming there are tuples with London as the city not in the USA.

If we are using the Consistency objective function, then adding a disjunctive condition does not increase the score. For example, adding

```
OR City = "London"
```

might increases the number of existing tuples produced by Θ_W. On the other hand, adding a conjunct such as

```
AND Country = "India"
```

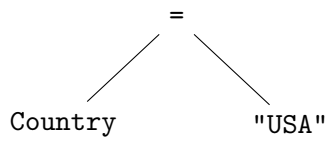

Fig. 3.6: Removing from WHERE clause

may reduce the number of tuples.

- Deleter - The Deleter mutator works by removing a conjunct or disjunct from Θ_W. Let Θ_W contain the following tree pattern (o, t_L, t_R), then the Deleter mutator applied to the left child yields t_R. As an example, suppose the WHERE clause includes the following comparison.

```
Country = "Mexico" OR Country = "USA"
```

Then we can remove the left condition from the clause updating it to the following.

```
Country = "USA"
```

The expression tree changes from Figure 3.2 to Figure 3.6.

The choice of which condition to remove from Θ_W is driven by the objective function and expression. If we are using the Consistency objective function, removing the most selective conjunct yields the maximal score. For example if Θ_W is

```
Country = "USA" AND City = "Chicago"
```

Then, removing `Country = "USA"` is best because the resulting Θ'_W is consistent with Θ_W. For a disjunction, removing the least selection condition is preferable. If Θ_W is

```
Country = "USA" OR Country = "Mexico"
```

and given the histogram in Figure 3.7 shows that

```
Country = "USA"
```

produces more tuples and therefore we should remove

```
Country = "Mexico"
```

for Θ'_W. Similarly, when Diversity objective function is chosen, and conjunct is present, removing the least selective is preferred. As an example, if Θ_W is

```
Country = "USA" AND City = "Chicago"
```

then

```
Country = "USA"
```

produces more tuples and therefore increases the Diversity score, whereas if the disjunct is present, then removing any of the conditions does not increase the Diversity score.

- Modifier - The Modifier mutator first applies a Deleter and then an Inserter, replacing an expression in a tree.

There are many potential patterns that could be applied. Histograms of the column values can be used to select the most efficacious. Suppose that the histograms shown in Figures 3.7-3.9 are available (query optimizers in DBMSs construct histograms to determine query selectivity). For each unique value, we look for values close to the percentage of tuples produced by Q with respect to the desired objective function. Suppose for instance that the query

```
SELECT *
FROM Customers
WHERE Country = "Mexico";
```

selects 6% tuples of the tuples in the relation. If we are using the consistency objective function then we want to modify the query in such a way as to produce a similar percentage of tuples. We can use the modifier pattern to replace the WHERE clause condition with a

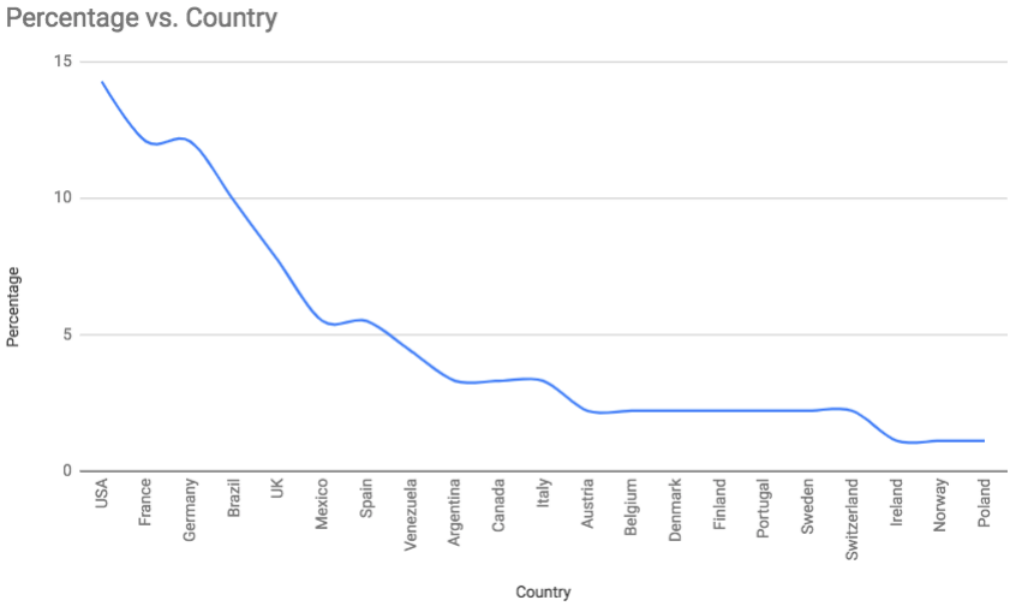

Fig. 3.7: Histogram of Country values showing the count of tuples for each value

clause with a similar selection cardinality. So, the following conditions would be acceptable alternatives.

- `Country = "Spain"`
- `Country = "UK"`
- `City = "London"`
- `City = "México D.F."`

3.1.2 Semi-join Transform

This is an extension to the WHERE clause transform to enable semi-join conditions to be added as a subquery. There are two cases for a semi-join: foriegn keys to a relation and foreign keys from a relation.

For foreign keys to a relation, we can extend a query with semi-joins to referenced relations. Suppose that we have the following query.

```
SELECT * FROM Orders;
```

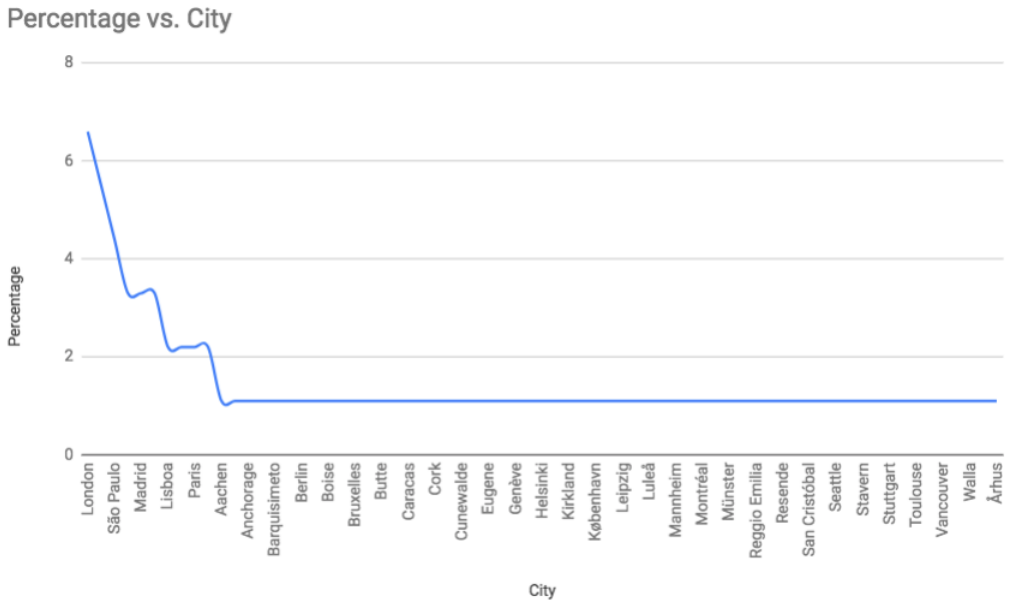

Fig. 3.8: Histogram of City values showing the count of tuples for each value

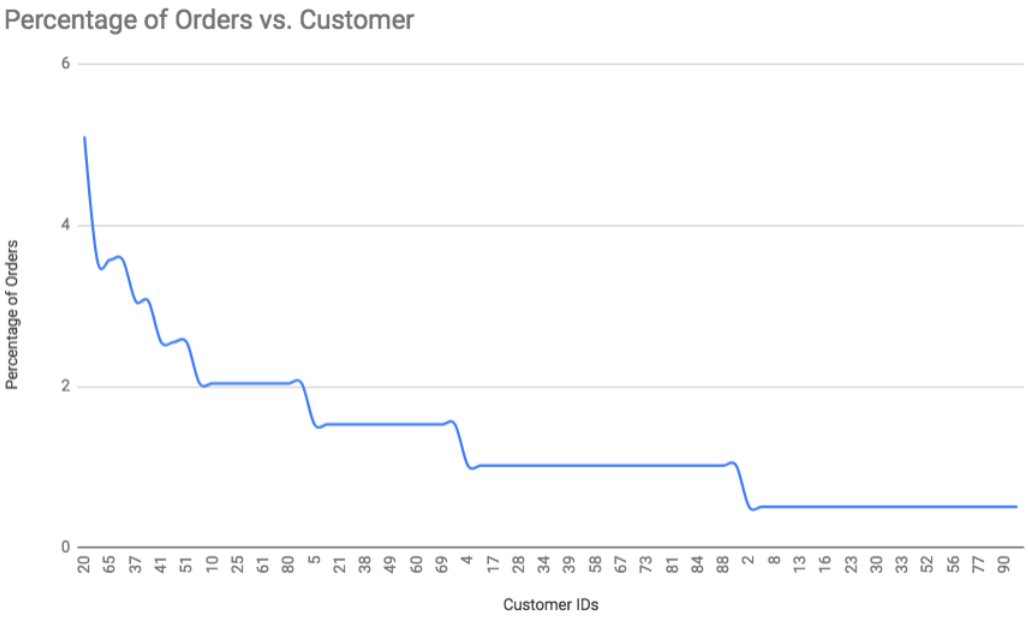

Fig. 3.9: Histogram of Orders values showing the count of tuples for each Customers

Then any foreign key to the `Orders` relation is a candidate for a semi-join. The columns which are foreign keys are `CID`, `EmployeeID`, and `ShipperID`. `CID` belongs to `Customers` relation, `EmployeeID` belongs to `Employees` relation, and `ShipperID` belongs to *Shippers* relation. Their schema's are described as follows respectively:

```
Customers(CID, Name, ContactName, Country, City, PostalCode)
Employees(EmployeeID, LastName, FirstName, BirthDate, Photo)
Shippers(ShipperID, ShipperName, Phone)
```

For every foreign key, we construct an histogram of values showing the count of tuples. The Column value which produces the most tuples is chosen. In our case `Country` USA, produces the most number of tuples and therefore it is added to Θ_W as, resulting in the following query.

```
SELECT *
FROM Orders
WHERE CID IN
   (SELECT CID
    FROM Customers
    WHERE Country = "USA");
```

For the `Employees` relation, `FirstName Michael`, produces the most number of tuples and therefore it is added to Θ_W as, resulting in the following query.

```
SELECT *
FROM Orders
WHERE EmployeeID IN
   (SELECT EmployeeID
    FROM Employees
    WHERE FirstName = "Michael");
```

For the `Shippers` relation, there is no column value that stands out and produces the most number of tuples, so we choose `ShipperID 1` to add it to Θ_W.

```
SELECT *
FROM Orders
WHERE ShipperID IN
   (SELECT ShipperID
    FROM Shippers
    WHERE Shippers = "1");
```

Foreign key relationships from the table are also candidates for a semi-join transform. In our case, `OrderDetails`, which has the following schema, is the only relation where `OID` is used as a foreign key.

`OrderDetails(OrderDetailID, OrderID, ProductID, Quantity)`

For every column in `OrderDetails` we construct an histogram of values showing the count of tuples. The value that produces the most tuples is chosen. For the Diversity Objective function, if there is already a condition present in the the original query, then the new condition uses a disjunct clause. However, if there is no already present condition, adding this condition does not give increase the Diversity score. For Consistency Objective function, using a conjunct is preferable. In our case `ProductID` 31, produces the most number of tuples and therefore it is added to Θ_W as, resulting in the following query.

```
SELECT *
FROM Orders
WHERE OrderID IN
   (SELECT OrderID
    FROM OrderDetails
    WHERE ProductID = 31);
```

3.1.3 Modify subquery Transform

We can also modify a subquery. There are several ways to modify a subquery because it could be correlated or non-correlated, scalar-producing, single-column or multi-column,

and the operator to compare the result of the subquery, *e.g.,*, `IN` or `EXISTS`. A correlated subquery is a subquery that uses values from the outer query. So, we do not consider modifying the correlated subquery, however, we consider changing the keyword connecting the subquery, for instance, `IN` to `NOT IN` or $>$ to $<$. For a query containing a non-correlated subquery, we look at what type of what type of result does the subquery produce, is it single column, multi-column or scalar value producing? If it is scalar producing, we do nothing, because we can not improve it. Otherwise we modify the subquery it the same way, by changing the keyword connecting the subquery.

3.1.4 Remove subquery Transform

Another approach to modify a query containing a sub query is to remove the subquery. If the initial query is the following.

```
SELECT *
FROM Customers
WHERE CID IN
   (SELECT CID
    FROM Orders);
```

Then, removing the existing subquery condition updates it to the following query.

```
SELECT *
FROM Customers;
```

Modifying a query by this transformation may or may not increase the count of tuples produced by the original query depending on whether a conjunct or disjunct is present. For the diversity Objective function, if a subquery is the only condition present, then the measure is increased by removing the subquery. If more than one condition is present, then it depends on the expression. For example, for an query like the following

```
SELECT *
FROM Customers
```

```
WHERE County = "Mexico"
OR CID IN
   (SELECT CID
    FROM Orders);
```

where the subquery is a disjunct clause, removing the subquery condition decreases the count of tuples, reducing the diversity score. However, for a query like

```
SELECT *
FROM Customers
WHERE County = "Mexico"
AND CID IN
   (SELECT CID
    FROM Orders);
```

where the subquery is joined by a conjunct, removing the subquery condition may or may not increase the count of tuples, increasing the diversity score. For the consistency objective function, depending on the subquery condition, the count of tuples may increase or decrease.

3.2 Aggregate Transform

The aggregate transformation modifies an aggregate in a query.

Definition 3.2.1 *The Aggregate Transform, $\mathcal{A}$, transforms the list of grouping columns, the list of projected columns, and the list of tables containing data as follows.*

$$\mathcal{A}(Q(\overline{C}, \overline{T}, \overline{G}, \Theta_H, \Theta_W), D) = Q(\overline{C}', \overline{T}', \overline{G}', \Theta_H, \Theta_W)$$

■

QAA uses the flowchart in Figure 3.10 to determine how to apply the Aggregate transform. First QAA checks a query for the presence of an aggregate, if it is present, then it can be either removed or modified by adding or removing columns from the list of grouping columns in $\overline{G}$.

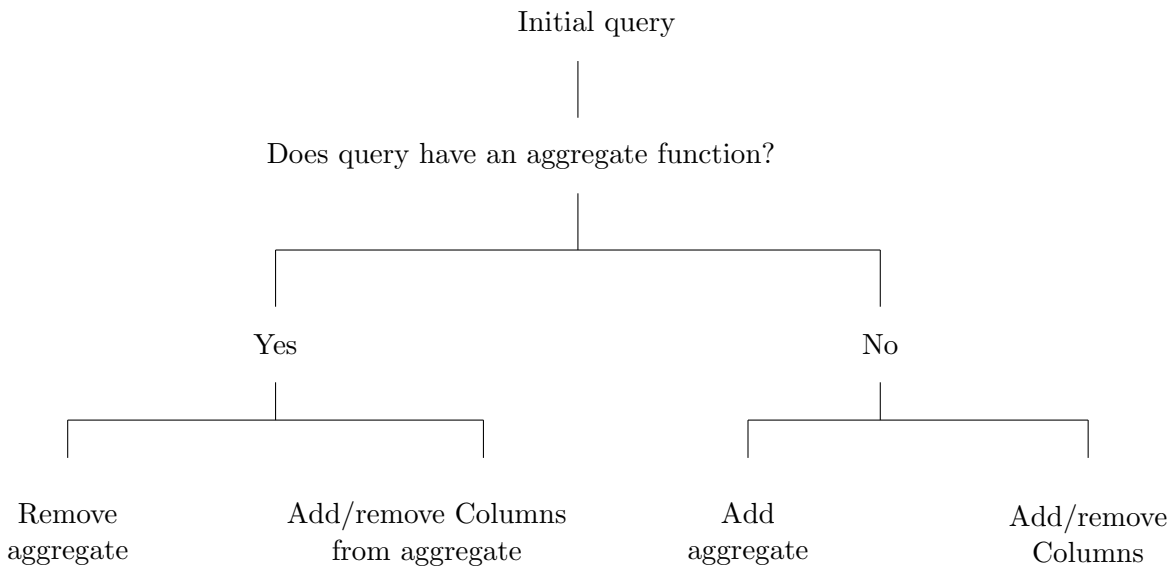

Fig. 3.10: Selection Processing Tree

3.2.1 Remove Aggregate

For a query with $\overline{G}$, a list of grouping columns present as $g_1, \ldots, g_k$, it can be modified by removing the `GROUP BY` clause. Removing the clause also changes the list of projected columns $\overline{C}$. For instance, given the following query:

```
SELECT Count(*), City, Country
FROM Customers
GROUP BY City, Country;
```

removing the `GROUP BY` clause also leads us to remove `Count(*)`, one of the projected columns as it produces the count of tuples in every group. Among the list of columns projected $\overline{C}$, we try every possible combination and the final query is chosen based on the objective function chosen by the user. When the objective function is the Diversity objective function, the query producing the maximum number of distinct tuples is chosen from alternatives because it lets us maximize the diversity score. However, removing the `GROUP BY` clause does not increase in the diversity score. In our example, "City", "Country" and "City, Country" produces 76, 23 and 76 records respectively. and therefore, we see

that, in case of Diversity Objective function, it does not make sense to remove the `GROUP BY` clause.

For the Consistency objective function, among the different combinations of columns, the combination producing the least change in the number of tuples is chosen. For example, suppose that the original query produces 76 tuples therefore

```
SELECT City, Country
FROM Customers;
```

is chosen because it produces 76 tuples, *i.e.,*, no change in the number of tuples produced.

3.2.2 Add/remove Columns to Aggregate

For queries with an aggregate function, we transform it by either inserting or removing columns from the list of `GROUP BY` columns. This transformation may result in change of projection of columns and the list of tables in the query. There are several cases.

- Adding a Column to $\overline{G}$ - We modify a query by adding more columns to the `GROUP BY`. From the list of tables in the `FROM` clause, we try adding columns to the `GROUP BY` clause. For the diversity objective function, suppose the query is the following.

  ```
  SELECT Count (*), Country
  FROM Customers
  GROUP BY Country;
  ```

  and it produces 23 distinct results. The `Customers` relation can be joined to other relations as follows.

  - Customers
  - Customers JOIN Orders
  - Customers JOIN Customers
  - Customers JOIN (Customers JOIN Orders)
  - Customers JOIN (Orders JOIN Orders)

Table 3.1: Customers Relation.

List of Columns	*Tuples produced*
Country, CustomerID	100
Country, CustomerName	100
Country, Address	98
Country, ContactName	97
Country, PostalCode	95
Country, City	76

Table 3.2: Customers JOIN Orders Relation.

List of Columns	*Tuples produced*
Country, OrderID	400
Country, CustomerID	81
Country, EmployeeID	103
Country, OrderDate	293
Country, ShipperID	51

We proceed by adding columns to the existing $\overline{G}$ list, we get the results shown in Table 3.1.

The following query is chosen as the best option.

```
SELECT Count (*), c.Country, o.OrderID
FROM Customers as c JOIN Orders as o
WHERE c.CustomerID = o.CustomerID
GROUP BY c.Country, o.OrderID;
```

For the consistency objective function QAA chooses differently. The following query is the best option.

```
SELECT Count (*), c.Country, o.ShipperID
FROM Customers as c JOIN Orders as o
WHERE c.CustomerID = o.CustomerID
GROUP BY c.Country, o.ShipperID;
```

- Removing a column from the `GROUP BY` clause - A column or columns can also be removed. For the list of Columns already present in $\overline{G}$, we remove one at a time and count the number of DISTINCT tuples produced by each resulting list of columns. Suppose our query originally is the following, and produces 76 tuples.

```
SELECT Count(*), City, Country
FROM Customers
GROUP BY City, Country;
```

  Then removing both columns from the `GROUP BY` produces 23 and 76 tuples respectively. For the Consistency Objective function, we would choose the following query as the enhanced query since it also produces 76 tuples.

```
SELECT Count(*), City
FROM Customers
GROUP BY City;
```

  For the Diversity objective function, removing a column from the `GROUP BY` clause does not increase the score.

3.2.3 Add Aggregate

To add an aggregate, QAA looks at $\overline{C}$, the list of projected columns. It adds an aggregate for one of the columns. For the Consistency objective function adding an aggregate on existing columns does not add new tuples to the query and therefore any aggregate added scores the same. For the Diversity objective function adding an aggregate does increase the score. For example consider the following query.

```
SELECT City, Country
FROM Customers;
```

Assume the query produces 76 distinct tuples. For every combination of columns present, *i.e.,* `City`, `Country`, and `Country, City`, we add an aggregate. These new queries end

up producing 76, 23 and 76 results. Among the three, the query with the most projected columns producing a similar count of tuples is chosen.

```
SELECT Count(*), City, Country
FROM Customers
GROUP BY Country, City;
```

3.3 Add/remove Columns

Definition 3.3.1 *The Add/remove Columns transform, $\mathcal{C}$, transforms the list of columns and the list of tables containing data as follows.*

$$\mathcal{C}(Q(\overline{C}, \overline{T}, \overline{G}, \Theta_H, \Theta_W), D) = Q(\overline{C}', \overline{T}', \overline{G}, \Theta_H, \Theta_W)$$

■

A query can be modified either by adding or removing a column from the list of projected columns. The transformation has several cases.

- **Removing a Column** - For a list of Columns already present as $c_1, \ldots, c_n$ we remove one column and count the number of DISTINCT tuples produced. Suppose that the following query produces 76 tuples.

  ```
  SELECT City, Country
  FROM Customers;
  ```

  We remove `City` and `Country` one at a time resulting in queries Q^1

  ```
  SELECT Country FROM Customers;
  ```

  and Q^2.

  ```
  SELECT City FROM Customers;
  ```

Table 3.3: Number of tuples per list of columns

List of Columns left	*Tuples produced*
Country	23
City	76

Table 3.4: Adding Columns

List of Columns	*Tuples produced*
Country, City, Customers.*	100
Country, City, Orders.*	30400
Country, City, Orders.*, CustomerID.*	7600

The number of distinct tuples is shown in Table 3.3.

- **Adding a Column** - Increasing $c_1, \ldots, c_n$ to $c_1, \ldots, c_{n+1}$ changes the schema. To add a column, we start by looking at a relation already present in a query. All columns are added to the list and the DISTINCT number of tuples produced are accounted. suppose the following query produces 76 tuples.

```
SELECT City, Country
FROM Customers;
```

Then QAA considers the modifications shown in Table 3.4. For the Diversity objective function, the combination producing the most tuples is chosen and yields the following enhanced query.

```
SELECT City, Country, Orders.*
FROM Customers JOIN Orders
WHERE Customers.CID = Orders.CID;
```

For the Consistency objective function, the most similar producing query is chosen.

```
SELECT City FROM Customers;
```

3.4 HAVING clause Transform

The `HAVING` clause is a predicate for groups. QAA can also transform this clause.

Definition 3.4.1 *The HAVING clause transform, $\mathcal{H}$, transforms the HAVING clause predicate as follows.*

$$\mathcal{H}(Q(\overline{C}, \overline{T}, \overline{G}, \Theta_H, \Theta_W), D) = Q(\overline{C}, \overline{T}, \overline{G}, \Theta'_H, \Theta_W)$$

■

The transformation cases are illustrated in Figure 3.11. As with the WHERE clause transform, the predicate can be modified by using an inserter, deleter or modifier pattern. The transformation depends on whether the query already has a `HAVING` clause.

- Since the `HAVING` clause applies to groups, it can only be added in situations where the query already has a `GROUP BY` clause. For every column present, we construct a histogram for all values and the count of tuples produced by them. The column and value with the maximum score is selected and is added to Θ_H.

- When the query already has a `HAVING` clause, it can be modified in the similar manner as `WHERE` clause transform. For an Inserter pattern, the choice of which conjunct and condition to insert to Θ_H is driven by the objective function. Adding a disjunct for the Diversity objective function is preferred, however, both conjuncts and disjuncts produces different results for the Consistency objective function, depending on the condition added. For the Deleter pattern, the choice of which condition to remove from Θ_H is driven by the objective function and the conjuncts. For a Diversity objective function, removal of the least selective conjunct is preferred. For the Consistency objective function, removal of most selective conjunct or the least selective disjunct is preferred.

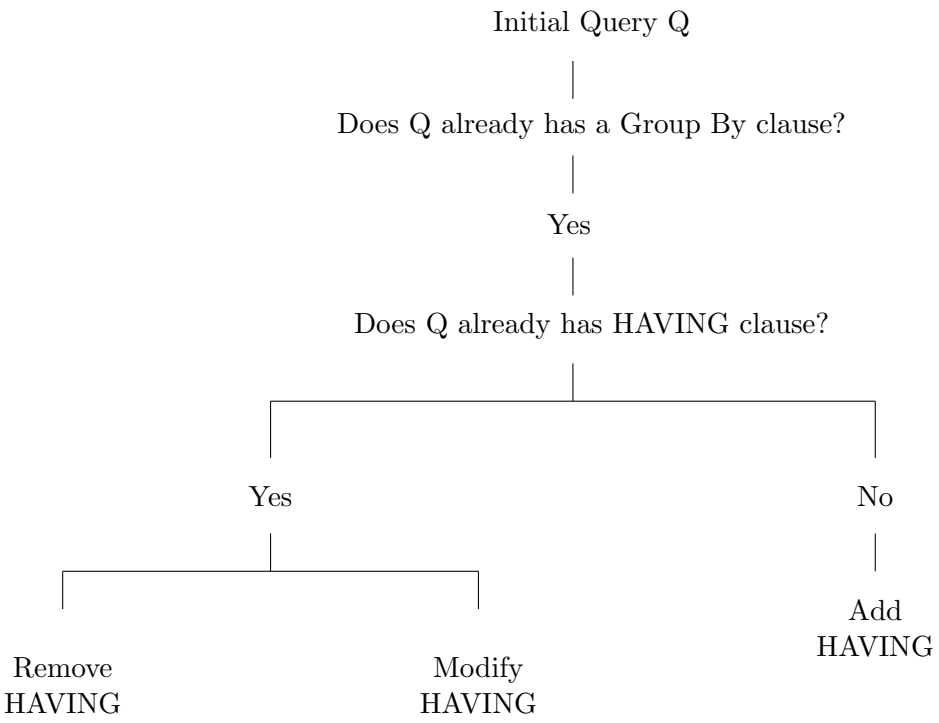

Fig. 3.11: HAVING condition Processing Tree

3.5 Limitations of Query AutoAwesome

The space of potential query enhancements is large. To make it possible to explore a small part of that space we did not consider the following potential enhancements.

1. Basic SQL-92 - SQL has grown from a small language to one encompassing many different aspects, such as temporal queries. We limited QAA to a basic SELECT statement in SQL.

2. `ORDER BY` clause - Tuples in a relation do not have a specific order, but they can be ordered in the result of a query. However, the ordering only changes the presentation of the results, so we did not consider enhancing the `ORDER BY` clause. Similarly we did not consider the `LIMIT` clause.

3. String comparison functions such as `LIKE` and wildcard characters, and inequality comparisons - We use histograms for different values and their frequencies to estimate the number of tuples produced by a query. Modifications of the `WHERE` and `HAVING`

predicates involve comparison operations. We limited the number of comparison operations, excluding string comparison functions and wildcard characters. Similarly, we did not consider inequality comparison operators such as $<$.

4. Relational operations - Relational operations such as relational union, intersection and difference are used to combine or exclude tuples with same schema from two or more relations. In QAA algorithm, we look at modifying queries based on its different components rather than adding or removing additional relations.

5. Aggregate functions - We limited aggregates to `COUNT` and did not consider `MAX`, `MIN`, `AVG`, `SUM`, or windowing functions.

6. Duplicate elimination - Adding the `DISTINCT` keyword to the query eliminates duplicates from the result. We did not consider duplicate elimination as it gives essentially the same query.

7. Multicolumn producing subqueries - The `EXISTS` and `NOT EXISTS` keywords are used to test whether a subquery produces a result. Though necessary to relational division and some operations, we did not see potential enhancements using these keywords.

8. Correlated subqueries - Queries with correlated subqueries are difficult to enhance since the inner and outer queries are dependent. We limited QAA to enhancing only non-correlated subqueries.

CHAPTER 4

Examples of Query AutoAwesome

In this chapter we present an example of QAA. Assume that our initial query is the following and produces six tuples.

```
SELECT *
FROM Customers
WHERE Country = "Mexico";
```

The following transformations can be applied to generate enhanced queries, as there is no subquery, aggregate function or `HAVING` clause.

- WHERE clause transform - On applying the WHERE clause transform, removing the existing WHERE condition produces a query with greatest diversity score which is, 100 - 6 = 94, yielding the following enhanced query.

  ```
  SELECT * FROM Customers;
  ```

  For the Consistency objective function, adding a condition using an Iserter mutator with a disjunct clause makes sense, as it produces only a small change. Any of the following disjuncts add at most one tuple to the result.

  ```
  Country = "Ireland";
  ```

  ```
  Country = "Poland";
  ```

  ```
  Country = "Norway";
  ```

  ```
  City = "Vancouver";
  ```

However, by using a Modifier mutator, we can look for literals that produces the same number of tuples as the original query and thus a better consistency score, as following.

```
Country = "Spain"

Country = "UK"

City = "London"

City = "México D.F."
```

We can select any of the above which yields the following enhanced query.

```
SELECT *
FROM Customers
WHERE Country = "Spain";
```

- SEMI JOIN transform - For this transformation, we look at the foreign key relationships for `Customers`. The primary key `CID` is a foreign key in the `Orders` relation. For every column in `Orders`, we construct a histogram for distinct values and their counts. We see that `ShipperID 2` produces the most number of tuples *i.e.,* 181. For the Diversity objective function, we add the subquery condition with a disjunct resulting in the following query.

```
SELECT *
FROM Customers
WHERE Country = "Mexico"
OR CID IN
   (SELECT CID
    FROM Orders
    WHERE ShipperID = 2);
```

For the Consistency objective function, we add a conjunct, modifying the final queries to the following.

```
SELECT *
FROM Customers
WHERE Country = "Mexico"
AND CID IN
   (SELECT CID
    FROM Orders
    WHERE ShipperID = 2);
```

- Add Aggregate transform - For $\overline{C}$, we try every combination and find that grouping by any of `Name`, `ContactName`, and `PostalCode` produces the same number of tuples. This transformation is only valid for the Consistency objective function, modifying the query as follows.

```
SELECT *, Count(*)
FROM Customers
WHERE Country = "Mexico"
GROUP BY PostalCode;
```

- Add/Remove columns - For the Consistency objective function, while removing a column from $\overline{C}$, we look for similar number of tuples produced. In our case, removing any of the columns produces 6 tuples, leaving us with the following query.

```
SELECT CID, CustomerName,
    ContactName, Address, City, PostalCode
FROM Customers
WHERE Country = "Mexico";
```

For the Diversity objective function, adding a column to $\overline{C}$ and the combination producing most number is chosen. In our case, the following query scores highest.

```
SELECT City, Country, Orders.*
FROM Customers JOIN Orders
WHERE Customers.CID = Orders.CID
AND Customers.Country = "Mexico";
```

In the next round the enhanced queries produced above become input to the QAA algorithm. Objective function scores are produced with respect to the original query. Consider the following query.

```
SELECT * FROM Customers
WHERE Country = "Mexico"
OR CID IN
   (SELECT CID
    FROM Orders
    WHERE ShipperID = 2);
```

The following transformations can be applied to get new queries, as there is a subquery, but no aggregate function or `HAVING` clause.

- WHERE clause transform - On applying the WHERE clause transform, removing one of the existing WHERE conditions does not produce a query that increases diversity. Therefore, we end up adding one more disjunct. From the histograms, we find that `Country = "USA"` produces the most tuples and therefore it is added, modifying the query to the following.

```
SELECT *
FROM Customers
WHERE Country = "Mexico"
OR Country = "USA"
OR CID IN
   (SELECT CID
    FROM Orders
```

```
    WHERE ShipperID = 2);
```

This additional condition adds seven tuples to the query result.

- Modify subquery transform - For modifying a subquery already present, we see that the subquery is a single column producing, non-correlated subquery and therefore can be modified according to Query AutoAwesome algorithm by changing `IN` keyword to `NOT IN` as in the following query.

```
SELECT *
FROM Customers
WHERE Country = "Mexico"
OR CID NOT IN
   (SELECT CID
    FROM Orders
    WHERE ShipperID = 2);
```

This increases the diversity score, as `Customers` with `ShipperID 1` and `3` were included.

- Remove subquery transform - Modifying a query by removing a subquery may increase the diversity score. In our example, however, removing the subquery condition,

```
SELECT CID
FROM Orders
WHERE ShipperID = 2;
```

results in producing the original query again and therefore is discarded.

- Add/Remove columns - For the Diversity objective function, adding a column to $\overline{C}$ and the combination producing most number is chosen. The following query is best.

```
 SELECT c.City, c.Country, o.*
```

```
FROM Customers as c JOIN Orders as o
WHERE c.CID = o.CID
AND (c.Country = "Mexico"
OR c.CID IN
      (SELECT CID
       FROM Orders
       WHERE ShipperID = 2));
```

At this point we assume that the threshold objective function score has been exceeded and the algorithm stops producing enhanced queries, showing all of the above queries to the user, ranked by the objective function score.

CHAPTER 5

Related Work

The branches of renovation in the form of database and applications are widely focuses in this research, which is shown in Figure 5.1 [2]. Renovation affects both *database* and *applications* as well as the interaction between database and application. Further, renovation of *database* depends on *data* or *schema*, and renovation of *application* depends on *code* or *performance*.

With respect to *data*, it can be improved, either by enhancing the existing *queries* or by refining the quality of *data*. Renovation of the Schema can be achieved through data *migration* to a different DBMS, *refactoring* the schema or by *extension* of the known database to allow wider range of data. In order to provide renovation to the database side, efforts have been found in both data and schema, as:

Database/Data/Queries Nandi and Jagadish [8] display interactive query construction compilers that interacts with end-users to help them build a rich search query.

Database/Data/Quality Databases are often times found to have missing attribute values. To tackle this situation, different solutions have been suggested, *e.g.,* reformulating user query automatically based on mined correlations between database attributes. Based on functional dependencies and relationships present in relational databases, their data quality can be enhanced, *c.f.,* [5, 7] .

Database/Schema/Migration The presence of vast variety of DBMSs drove way for automating data migration [11] and to the development of migratiion tools, *c.f.,* [1].

Database/Schema/Extension An et al. [3] present a solution to evolving data gathering requirements by automatically mapping and integrating user created data entry forms into an existing database.

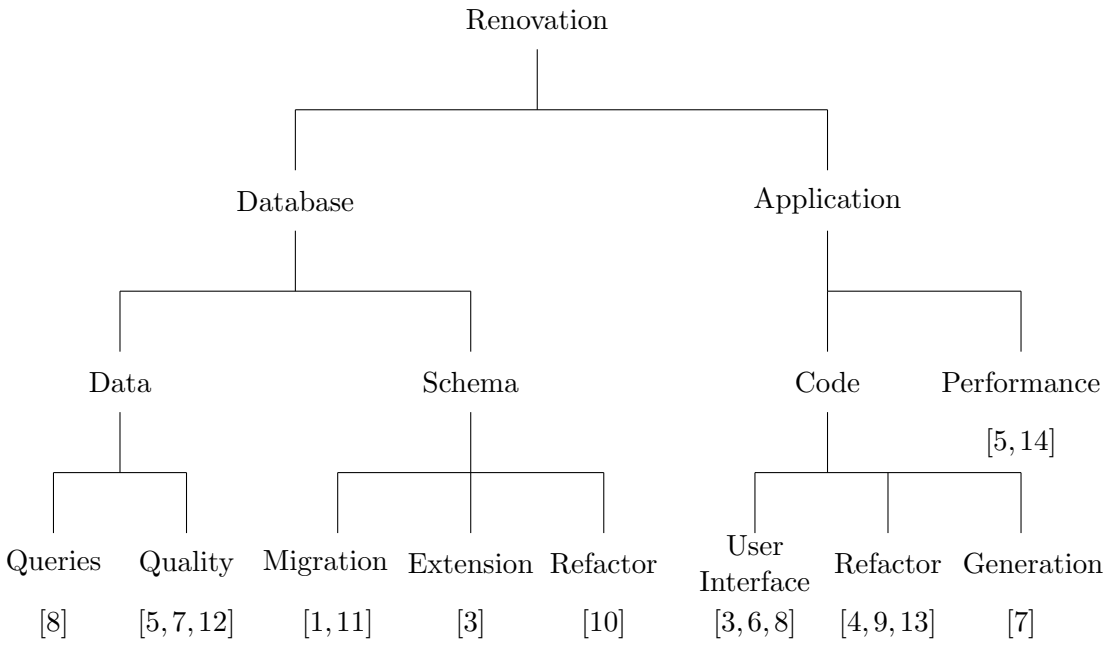

Fig. 5.1: Taxonomy of database renovation research

Database/Schema/Refactor Vial [10] talks about the importance of refactoring database schema, challenges and key lessons learned from their experience in database refactoring.

On the other hand, renovation on *application* side of taxonomy can be carried out without affecting the existing database. This improvement can be provided to DBApp by targeting *performance* or by making relevant changes to *code*. *Code* changes are done by *improving user interfaces*, by *refactoring* already existing code to improve basic structure and quality, or by *generating new code*. Previous works in the application side to improve DBApp are:

Application/Code/User Interface Jagadish et al. [6] studies databases with respect to usability. They introduce a set of pain points and ways to address them. Nandi and Jagadish [8] demonstrates an interface form with a single input text box interacting with the end-users as they type, guiding them with query building.

Application/Code/Refactor In order to improve the internal structure of software systems, Cedrim [4] introduces a novel model of machine learning techniques to recognize code refactoring opportunities. Sharma [9] presents another automated technique to address code smells such as long and incohesive methods, and duplicated code called extract-method refactoring. Xin et al. [13] created the Migration Evaluation and Enablement Tool called MEET DB2 to automatically provide detailed evaluation of database migration complexity.

Application/Code/Generation Jayapandian and Jagadish [7] present an automated technique to generate a reasonable set of forms that covers upto 60-90% of user queries using important relationships present in a database.

Application/Performance Zisman and Kramer [14] develops a way to ease up information discovery with the help of automatic indexes. Ilyas et al. [5] develops an algorithm to automatically discover correlations and soft functional dependencies between pair of columns.

Query AutoAwesome is in the area of **Database/Data/Queries**. We generate new queries from existing queries using both the data and the schema.

CHAPTER 6

Plans for Implementation

In this chapter we describe our plans for implementation. Figure 6.1 gives an overview of the implementation.

The first step to enhancing a query is to parse the query. For this step we plan to use ANTLR, and the ANTLR SQLite grammar. ANTLR generates a parser and semantic actions associated with the parse. For our purposes the parser will parse an SQL query, extracting the parts of the query pattern in $Q(\overline{C}, \overline{T}, \overline{G}, \Theta_H, \Theta_W)$. For example if the query is the following.

```
SELECT Count(*), City, Country
FROM Customers
WHERE Country = "USA"
GROUP BY City;
```

The parser will identify its different parts as follows.

- $\overline{C}$ = [City, Country]
- $\overline{T}$ = [Customers]
- $\overline{G}$ = [City]
- Θ_H = NULL,
- Θ_W = EQ(Country,"USA")

$\overline{C}$ is a list of projected columns, $\overline{T}$ is a list of tables that contain the data, $\overline{G}$ is a list of grouping columns, Θ_H is a predicate on the groups and Θ_W is a selection predicate. As there is no predicate on groups, it is NULL. The selection clause predicate is parsed into an expression tree for modification by the inserter, deleter, and modifier patterns. The bulk of

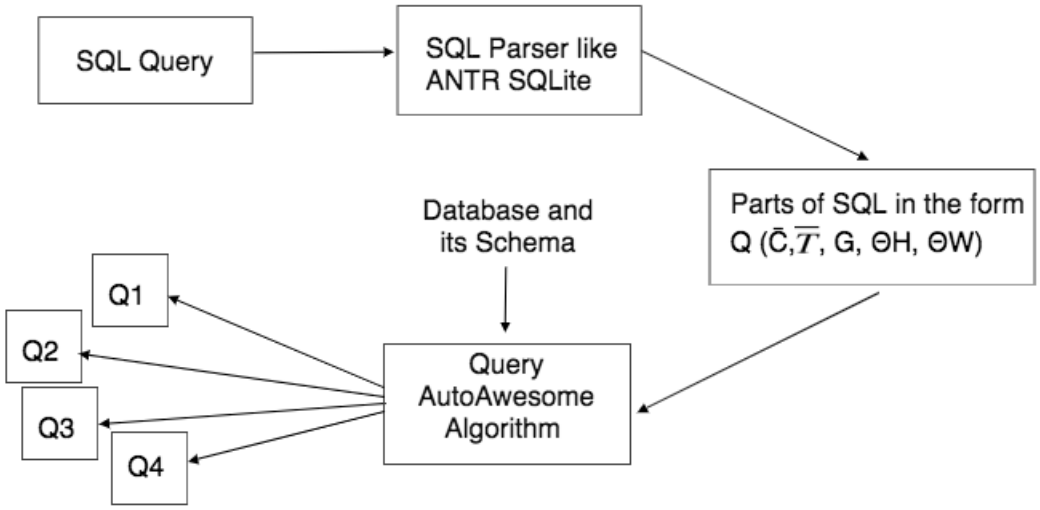

Fig. 6.1: Implementation

the transformation is done by semantic actions associated with parser rules. For instance, the parse rule for the SELECT clause will create the list of projected columns.

We anticipate that the parsing and generation of the enhanced queries will be fast. More costly will be the computing the objective functions. An important part of QAA is estimating or computing the tuples produced by a query. For the Diversity and Consistency objective functions without schema change, each enhanced query must be evaluated and the result compared to that of the original query. If schema change is present, only the count of tuples is needed. It is possible to obtain an estimate of the count from the query optimizer without running the query, but in general a large number of queries will have to be evaluated. While running a large number of queries will take a long time (hours to days) we do not anticipate that QAA needs to work quickly to achieve results. Google's AutoAwesome works offline for days to months after photos are uploaded. When ready, the AutoAwesomizer pushes enhanced photos to a client via e-mail.

CHAPTER 7

Conclusions and Future Work

Writing database queries needs expertise in SQL programming and demands a considerable amount of time and effort. In this thesis, we present an algorithm, called Query AutoAwesome, to automate SQL query enhancement. QAA works in a manner similar to Google's Auto Awesome tool. A query is enhanced by applying a technique from a suite of techniques, a user chooses to keep or discard the enhanced query. Query enhancement limits the level of knowledge, time and effort required to extend an SQL query.

QAA starts by ingesting a query, database schema, and the data. The query then is modeled as a query pattern with parameters presented in Chapter 2. These patterns and parameters are modified to generate new queries. Each new query is scored using an objective function and the top-k queries are chosen for further enhancements. The algorithm continues to enhance queries until some user given threshold is exceeded.

QAA can only enhance parts of SQL queries as discussed in Chapter **??**. Extended QAA to cover more of SQL is future work.

Another potential improvement is a GUI. Initially, the end user has to feed in the schema, data for the database and a preferred Objective function. Later, when they type in a valid SQL query, it will present them with alternative enhancements in a drop down menu. Each of these enhancements will be produced by making changes to different parameters of the initial query as discussed in Chapter 3, namely, WHERE clause transformation, Aggregate transformation, Add/remove columns transformation and HAVING clause transformation. Each of the above mentioned transformations can be present or absent depending on the structure of original query. For example, if the initial query does not have a GROUP BY clause, then HAVING clause transformation can not be applied to it. Each of these transformations can present more than one alternatives to let the user choose from. For instance, WHERE clause transformation can come up with more than

one column-literal combination that produces same objective function score. These Literals in a WHERE clause predicate can be presented to end user with a drop-down menu of alternatives. Graphical user interface lets user choose from different alternatives presented and therefore giving them the ability to choose, rather than presenting them the best modified query from the k-top based on objective function. As, despite of the same objective functions, user preferences can differ.

Another enhancement to Query AutoAwesome can be a made by conducting a user study. It focuses on understanding user expectations, behaviours and needs in order to ensure that the tool designed benefit end users by helping us align our tool with their core needs. Conducting user research can help us identify and address our own biases and misconceptions related to the project. In order to understand end user's need better, a user study can be conducted by presenting them with a questionnaire, to better know their needs and to compare them to what Query AutoAwesome algorithm does. End users will be presented with the original query and different modified versions. They will be able to choose based on their preference to know if the modified versions are actually enhanced queries.

REFERENCES

[1] SQL Server Migration Assistant.

[2] Jonathan Adams and Curtis E. Dyreson. Renovating database applications with dbautoawesome. In *Databases Theory and Applications - 29th Australasian Database Conference, ADC 2018, Gold Coast, QLD, Australia, May 24-27, 2018, Proceedings*, pages 94–106, 2018.

[3] Yuan An, Ritu Khare, Il-Yeol Song, and Xiaohua Hu. Automatically mapping and integrating multiple data entry forms into a database. In *Proceedings of ER*, pages 261–274, 2011.

[4] Diego Cedrim. Context-sensitive identification of refactoring opportunities. In *Proceedings of the International Conference on Software Engineering, ICSE*, pages 827–830, 2016.

[5] Ihab F. Ilyas, Volker Markl, Peter J. Haas, Paul G. Brown, and Ashraf Aboulnaga. Automatic relationship discovery in self-managing database systems. In *1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA*, pages 340–341, 2004.

[6] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 13–24, 2007.

[7] Magesh Jayapandian and H. V. Jagadish. Automated creation of a forms-based database query interface. *PVLDB*, 1(1):695–709, 2008.

[8] Arnab Nandi and H. V. Jagadish. Assisted querying using instant-response interfaces. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 1156–1158, 2007.

[9] Tushar Sharma. Identifying extract-method refactoring candidates automatically. In *Fifth Workshop on Refactoring Tools 2012, WRT '12, Rapperswil, Switzerland, June 1, 2012*, pages 50–53, 2012.

[10] Gregory Vial. Database refactoring: Lessons from the trenches. *IEEE Software*, 32(6):71–79, 2015.

[11] Guowei Wang, Zongpu Jia, and Manjun Xue. Data migration model and algorithm between heterogeneous databases based on web service. *JNW*, 9(11):3127–3134, 2014.

[12] Garrett Wolf, Aravind Kalavagattu, Hemal Khatri, Raju Balakrishnan, Bhaumik Chokshi, Jianchun Fan, Yi Chen, and Subbarao Kambhampati. Query processing over incomplete autonomous databases: query rewriting using learned data dependencies. *VLDB J.*, 18(5):1167–1190, 2009.

[13] Reynold Xin, Patrick Dantressangle, Sam Lightstone, William McLaren, Steve Schormann, and Maria Schwenger. MEET DB2: automated database migration evaluation. *PVLDB*, 3(2):1426–1434, 2010.

[14] Andrea Zisman. *Information discovery for interoperable autonomous database systems*. PhD thesis, Imperial College London, UK, 1998.