BROWN UNIVERSITY

Query Generator

Project Report

Ahsan Hussain

Advised by Stanley B. Zdonik

[This document describes the main components of the Query Generator project. There are some functional details but mostly it is a high level overview of our work. A section at the end is devoted to future work as well. For details on how to run the code, please see the attached Readme file. All code and documentation has been checked into a repository maintained by Alex Rasin. For questions or comments, please email ahsan@cs.brown.edu, qiao@cs.brown.edu or alexr@cs.brown.edu]

Table of Contents

Motivation

In order to study the performance of a physical database designer, we need to evaluate the quality of a design for a variety of input query sets. The conventional approach to this problem is to use a recognized query benchmark(s). Human designed benchmarks are generated explicitly for the design and database evaluation and come with a schema and a data generator. However, there are only a few well known and complete benchmarks available in the database community; and even then they may be tailored to a different setting (e.g. some are for OLAP others for OLTP). Therefore, there are simply not enough benchmarks to exhaustive test database design.

The next best thing is to acquire some "real" data, i.e. query and data set from a commercial company. Unfortunately, there are quite a few problems with real-world data. First of all, although the queries are real, the workload will only test a small part of the design space and might not predict the performance of other real-world set of queries. Secondly, even when the queries come with a data set, the amount of data is fixed as there is no data generator. Thus there is no good way to rerun similar experiments with a larger amount of data to observe the performance trends as the database grows larger (2x, 10x, etc). Finally, both the data and the queries are difficult to acquire since they contain sensitive commercial information; the data might require anonym zing before it can be used.

Using a query generator solves all of the problems outlined above. We can still start with an established benchmark (such as SSB) that comes with a peer-reviewed logical schema and a data generator that makes an effort to approximate a "real" data set. Building on that, we would like to be able to generate a wide variety of SQL queries to cover as much of the design problem space as possible and to show interesting trends in the output design as the input changes. After significant research, we were unable to find a SQL query generator that could generate queries based on a simple set of parameters (desired predicates, expected selectivity with support for random variables). As we will show in this document, such generator is invaluable for anyone analyzing the problems of physical design in OLAP setting. We will also show that even a relatively simple set of randomized parameters can cover a very large space of different query sets allowing us to observe important trends and evaluate physical designer performance. Testing against a query benchmark or a real workload has its place as a sanity check, but any such input corresponds to a single point in the large space of possible workloads.

Goals

The main goal of the Query Generator is to automatically generate a query workload based on input parameters. A set of input parameters defines a certain query set - verifying this input set lets us observe trends in the physical design sets. The Query Builder module generates queries on the fly: query parts (predicates or attributes) can be replaced by random variables.

The query parser does the reverse by extracting a set of inputs that have generated the parsed queries. The convenience of query structures is in that it is possible to apply transformations to the structure or consider the similarity metric between different structures.

The final goal of this project is to put together all of the components into a comprehensive tool that can parse a query set (a benchmark input, for example) and refine it by applying some simple transformations. In addition to simple testing of the query design space problem, we might want to find a certain answer, such as what kind of queries do very well and what queries will do poorly (given the data set). We should be able to continue refining the query set until we come to a stable answer.

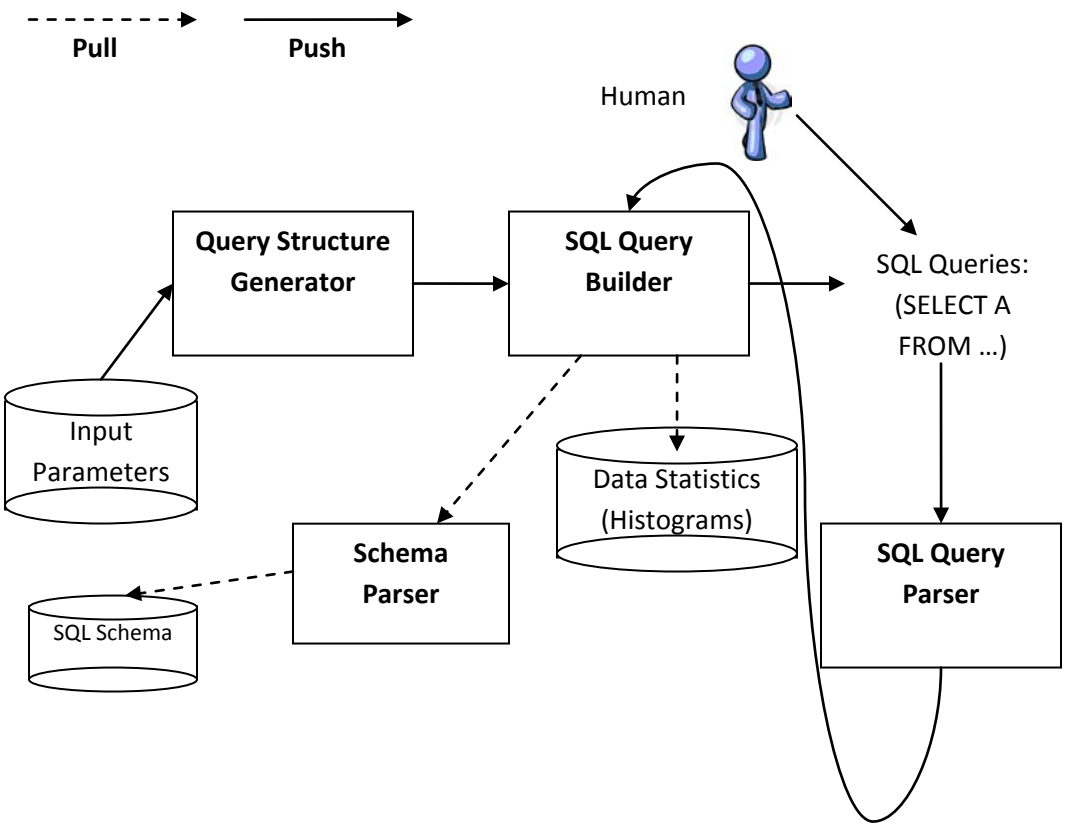
Main Components

The main components of the Query Generator are:

1. Data Statistics
2. Schema Parser
3. Query Structure Generator
4. SQL Query Builder
5. SQL Query Parser

These components are described in detail below.

Overall Architecture of Main Components

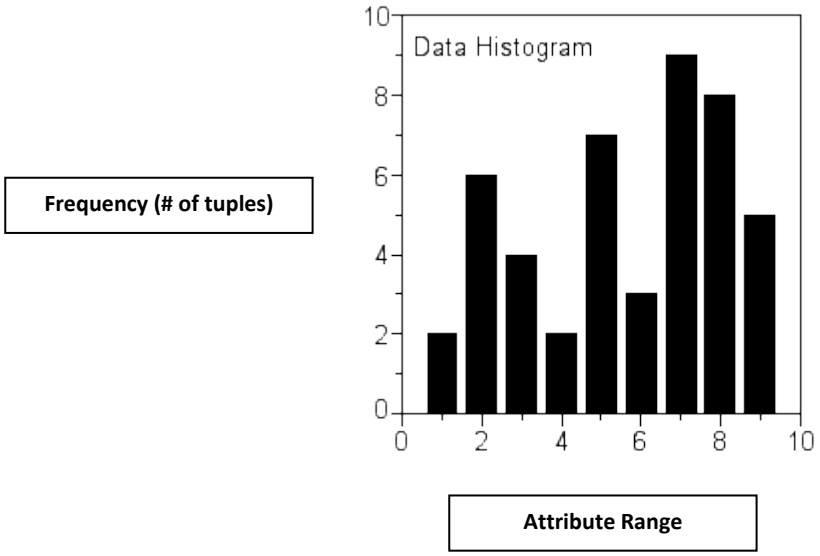


A Query Structure Generator takes in input parameters to generate multiple structures (per vector) which are then used by the SQL Query Builder (pulls information from Schema Parser and Data Statistics) to generate SQL queries. The queries generated by the SQL Query Builder as well as human-generated queries can then be passed into the SQL Query Parser to create original query structures (vectors).

1. Data Statistics

The statistics describe how data is organized within our database. This means that the queries we generate will only be as good as the granularity of our statistics. The more detailed our statistics, the greater accuracy we will have in our results.

For our project, we have a data statistics file for each of the five Star Schema tables (lineorder, customer, part, supplier, dwdate). There is a histogram (illustration shown below) per attribute for each of the tables mentioned above.



1.1 Example

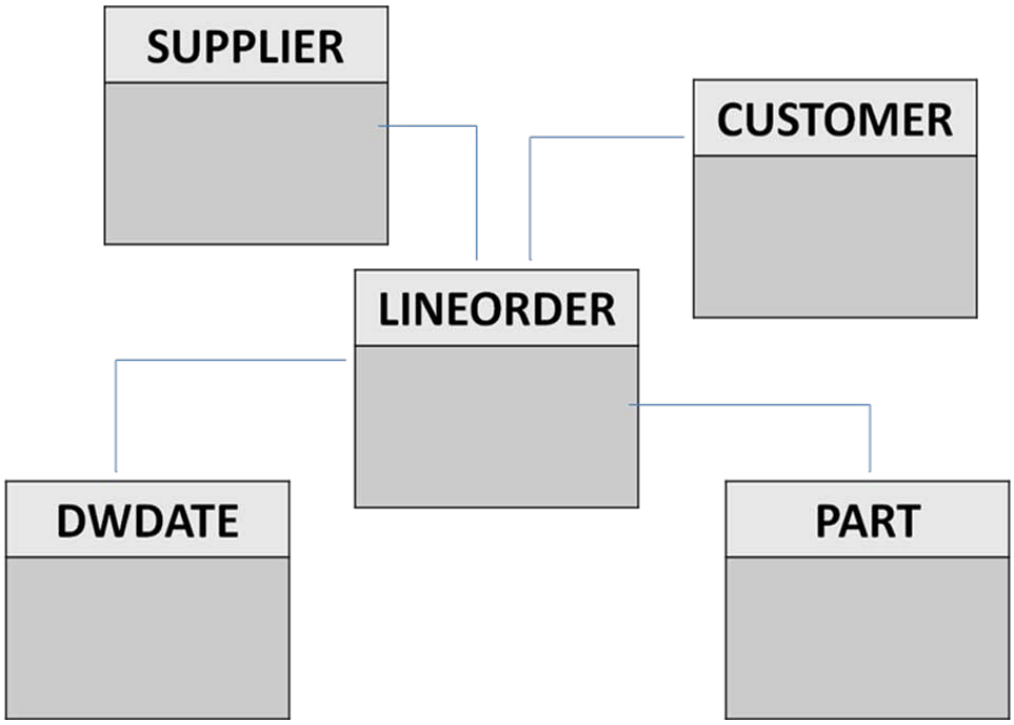
Here is what a typical statistics file looks like. All the attributes are specified on a frequency (number of tuples) and a range (or attribute) which may later be used by the different components of the Query Generator to either generate queries (from query structures) or to generate query structures (from queries).

```
AttrStart
customer.c_custkey integer
BETWEEN 1 and 120
120
BETWEEN 120 and 239
119
BETWEEN 239 and 358
119
BETWEEN 358 and 477
119
..
..
AttrEnd
AttrStart
customer.c_region varchar(12)
BETWEEN AFRICA
23890
BETWEEN AMERICA
24025
```

```
..
..
AttrEnd
..
..
```

2. Schema Parser

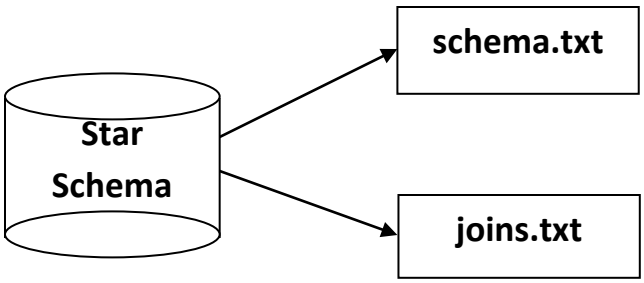
We use the Star / Snowflake Schema for this project. The general structure of the Star Schema is show below. We have one fact table in Lineorder and four dimension tables in Supplier, Dwdate, Part and Customer. The fact table contains foreign keys to all the dimension tables. The dimension tables each have only one primary key.



2.1 Functionality Details

This component of the Query Generator reads in Star Schema / Snowflake Schema, parses it and stores it in a data structure in memory and in text files. Later, during the actual query generation phase of Query Builder (see below), we refer to the schema (in memory and on disk) to make sure that we are building queries that are valid and conform to the schema.

The Star Schema is parsed to create two files: schema.txt and joins.txt



The file schema.txt stores Star Schema in a format that we can use to generate queries whereas the file joins.txt stores all the valid join keys. During query generation phase within the Query Builder, we refer to the schema.txt and joins.txt to make sure that the queries are joined on valid tables and that there is no violation of Star Schema.

We assume that there will only be one join key between any given pair of tables because we are using Star Schema.

3. Query Structure Generator

The Query Structure Generator generates query structures (described in detail below) based on a random distribution (uniform or normal) that may be later used by the Query Builder to generate queries.

3.1 Functionality Details

If a query structure specifies a predicate [dwdate.d_year: U, 0.1, 0.6], then we generate a selectivity based on uniform distribution [for example, dwdate.d_year: 0.232223222] within the range specified. We use a random seed to generator a random selectivity. This is done in order to make sure that the distribution is truly uniform.

The input to the Query Structure Generator is a set of query structures (vectors) and the output is also a set of query structures (vectors) but since there is a 1 to n mapping between input and output, we may see multiple vectors for each vector input.

3.2 Query Structure Details

This specifies the number of queries that will be generated for this query structure.

```
Num: 1
```

The Table tag specifies that tables that need to be joined in order to generate this query (or set of queries).

```
Table
dwdate, lineorder
```

The Selectivity tag specifies the attributes whose selectivity we need to take into consideration when generating this query (or set of queries). We can specify multiple attributes and but remember that the selectivity is calculated independently for each of the attributes.

```
Selectivity
dwdate.d_year: U, 0.1, 0.5
```

Both these formats are acceptable:

<table.attribute><:><uniform distribution><min><max>
<table.attribute><:><normal distribution><mean ><variance>

The Target specifies the attribute to select. In this case, we select all attributes from the dwdate and lineorder:

```
Target
*
```

3.3 Example

Here is what an input to the Query Structure Generator might look like:

```
Num: 20
Table
dwdate, lineorder, part
Selectivity
part.p_name: U, 0.1, 0.4
dwdate.d_yearmonth: U, 0.3, 0.5
Target
part.p_name

Num: 100
Table
lineorder, dwdate
Selectivity
lineorder.lo_orderkey: U, 0.2, 0.5
Target
*
..
..
```

INPUT

The Query Structure Generator generates twenty structures (vectors) each with predicate selectivity (uniform) between 0.1 and 0.4 on p_name attribute and between 0.3 and 0.5 on d_yearmonth attribute (in case of the first structure) and hundred query structures (vectors) each with selectivity (uniform) between 0.2 and 0.5 on lo_orderkey attribute.

The output of the Query Structure Generator for the first structure would generate twenty structures similar to:

OUTPUT

```
Num: 20
Table
dwdate, lineorder, part
Selectivity
part.p_name: 0.1891245671
dwdate.d_yearmonth: 0.3452376512
Target
part.p_name

Num: 20
Table
dwdate, lineorder, part
Selectivity
part.p_name: 0.2972456137
dwdate.d_yearmonth: 0.4529877129
Target
part.p_name

Num: 20
Table
dwdate, lineorder, part
Selectivity
part.p_name: 0.3328677431
dwdate.d_yearmonth: 0.3122385677
Target
part.p_name
..
..
```

The output for the second query structure would include hundred more query structures (vectors) similar to:

```
Num: 100
Table
lineorder, dwdate
Selectivity
lineorder.lo_orderkey: 0.2189768991
Target
*

Num: 100
Table
lineorder, dwdate
Selectivity
lineorder.lo_orderkey: 0.4898882817
Target
*

Num: 100
Table
lineorder, dwdate
Selectivity
lineorder.lo_orderkey: 0.2911223938
Target
*

..
..
```

OUTPUT

4. SQL Query Builder

The SQL Query Builder takes in a query structure or a set of query structures (generated by the Query Structure Generator) and generates corresponding SQL queries. For this process, the SQL Query Builder pulls information from the Data Statistics (Histograms) as well as the Schema Parser information stored within memory and on disk (in files). The information from Schema Parser is used to validate SQL queries generated by the SQL Query Builder and to add join keys to the SQL Query (in case of a join query). The interaction between SQL Query Builder and the Data Statistics is described in detail below when we discuss attribute selectivity.

4.1 Functionality Details

Below we provide a description of the way we compute selectivity on attributes. There are two ways to calculate selectivity. The difference between the two approaches is that one computes the selectivity in a way that could possibly return discontinuous ranges whereas the other only returns a single continuous range. This is a tradeoff in terms of accuracy in the case where we generate a single continuous range but we still want to give the user the option to be able to get back queries with a single range on an attribute.

4.1.1 Computing attribute selectivity

The main challenge was to be able to spit out the ranges that correspond to the given selectivity for each attribute (within the query structure). Here we used a brute force approach to try every possible combination of selectivity of each and every range in order to compute either an exact match or the best possible answer. This approach has a drawback in the time it takes to compute. In order to overcome this hurdle, we've added a way to relax this requirement (see Relaxation Option below) but the match may not be as accurate.

Here is a brief description of how our algorithm works to compute selectivity on an attribute. For details please refer to /Repository/Code/QueryGen.java.

```
findRange(double goal, double[] ranges, int size) {

        for each element in ranges {

                if(Math.abs(ranges[i]-goal)<0.001)  {//if the difference of current best value and

                // the desired value is less than 0.001,we assume that they are equal and

                // will return true to indicate the desired value is found.

                // Chaging the "0.001" to a bigger value can be viewed as a relaxation.

                        /*answerStack stores values that add up to current best answer */

                                answerStack.add(ranges[i]);

                                /* goalDiff stores the current difference from desired value */
```

```
                    goalDiff = 0; //goalDiff stores the difference between current best
                    // answer and the desired value
                            return true;
                 }
        }
        for each element in ranges {//if the desired value is not found, select a value and push
        // it in the current answer stack
                answerStack.add(ranges[i]); //push the selected value to answer stack
                newGoal = goal - ranges[i]; //set the new desired value
                if(newGoal < 0 && (-newGoal)<goalDiff)
                        goalDiff = -newGoal;
                        double[] newRanges = ranges remove ranges[i] and elements bigger than
                        newGoal; //construct the new ranges array for recursive call
                if(newRanges is empty){//no value is eligible for the new ranges array
                        answerStack.pop(); //pop out the selected value
                        continue;
                }
                else {
                        findRange(newGoal, newRanges, newSize);
                }
        }
        return false; //all combinations have been tested and the desired value is not found
}
```

4.1.2 Computing attribute selectivity using the continuous flag

The continuous flag (if specified by the user) uses the same approach as described above but only outputs SQL queries that conform to a single continuous range. We understand that this might result in a loss of accuracy on selectivity but it gives the user an option to only see queries that have a single attribute range.

Here is a brief description of how our algorithm works to compute selectivity using the continuous flag. Again for details please refer to /Repository/Code/QueryGen.java.

```
findRange_con(double goal, double[] ranges, int size){
for each element in the value array, denote as range[i]
    for each element after range[i], denote as range[j]
       compute: sum = range[i]+range[j] //start adding up values from range[i+1] until the sum
                                        // is bigger than the desired value
if(sum>goal){
//the best sum we can get for starting point range[i] is either at current position
//or the one before it

best_ans_start = i; //start index of answer range
goalDiff = min{(sum-goal), (goal-sum+ranges_left[j])} //choose the smaller one
if(goalDiff == sum-goal)
best_ans_end = j; //set end index of answer to be the current position
else
best_ans_end = j-1; // set end index of answer to be the one before current position
break;}
if(goalDiff < 0.001)
return true;
else
return false;
}
```

4.1.3 Relaxation Option

In order to reduce the level of computation when we determine a range that reflects the selectivity of an attribute, we have incorporated a “relaxation” option. This means that instead of finding an exact or a near exact match between a given selectivity and its range (on an attribute), we only compute a range that is a certain distance away from it (instead of being exactly equal to it). For example, if our algorithm is computing attribute ranges that fit within a selectivity of 0.4005, then instead of looking to find a combination that adds exactly to 0.4005, the relaxation allows the algorithm to return back a selectivity of 0.4000 or 0.4010 (±0.005 within the desired selectivity). For details on this, please refer to findRange() method within the RangeFinder.java in our project code within the repository. This results in a loss of accuracy when computing ranges but provides added functionality that may be of use when computation time is more important that the level of granularity.

4.2 Example

This is a simple query structure (vector) and its corresponding SQL Query we generate:

Input

```
Num: 1
Table
dwdate, lineorder
Selectivity
dwdate.d_year: 0.1198656667
Target
*
```

Here is the query that the Query Builder spits out:

Output

```
select *
from dwdate, lineorder
where (
dwdate.d_year = 1993 OR
dwdate.d_year = 1996)
AND
lineorder.lo_revenue = dwdate.d_datekey
```

5. SQL Query Parser

This module takes in an SQL query or a set of SQL queries and generates query structures (vectors). We generate one vector per query. The user can write a set of queries they want to parse within query.txt where each query is separated by a newline character. After we are done generating vectors, we store them in within vector.txt. Please note that there is a one to one relationship between a query and its query structure (vector).

5.1 Functionality Details

The Query Parser simply looks at the range (on an attribute) specified in a given query and computes a selectivity that reflects that range. This is somewhat opposite to what we achieve with the Query Builder.

5.2 Example

Here are a few SQL queries and the query structures (vectors) we generate for each. The queries are somewhat similar but the point is to show the different ways we are able to parse queries.

Here is an input SQL query to the SQL Query Parser:

```
select SUM(lo_revenue), d_year
from dwdate
where d_year between 1993 and 1996;
```

Input

The output query structure (vector) generated by the SQL Query Parser:

```
Num: 1
Table
dwdate
Selectivity
dwdate.d_year: 0.0122111223
Target
sum(lineorder.lo_revenue), dwdate.d_year
```

Output

Another SQL query as input:

```
select SUM(lo_revenue), d_year
from dwdate
where (d_year = 1995) or (d_year = 1996);
```

Input

The output from the SQL Query Parser:

```
Num: 1
Table
dwdate
Selectivity
dwdate.d_year: 0.1129874197
Target
sum(lineorder.lo_revenue), dwdate.d_year
```

Output

Another input SQL query:

```
select SUM(lo_revenue), d_year
from dwdate
where (d_year between 1992 and 1995) or (d_year between 1996 and 1998);
```

Input

A corresponding output query structure (vector):

```
Num: 1
Table
dwdate
Selectivity
dwdate.d_year: 0.4918634954
Target
sum(lineorder.lo_revenue), dwdate.d_year
```

Output

Support Documentation

All supporting documentation (including a Readme file) and project code are found within the repository maintained by Alex Rasin (alexr@cs.brown.edu). Please refer to the Readme file for details on how to run project code. Input query structures (vectors) and SQL queries are entered in specified files and output is generated within specified files as well (see Readme for details).

Future Work

There is some functionality that should be implemented in future versions of Query Generator. We will briefly outline those here:

- We want to have the ability to apply selectivity transformations to predicates such as doubling the selectivity on an attribute or scaling it by a factor.
- We do not do any error checking in case the user species are query structure that does not conform to the Star Schema (Schema Parser)
- We want to be able to generate queries based on normal distribution (Query Builder)
- Ability to pick a random attribute to apply selectivity on (Query Builder)