VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

A Tool for Generating Data Collections, Queries and Statistical Functions

Nástroj pro generování datových kolekcí, dotazů a statistické účely

Bogdan Postolov

Bachelor Thesis

Supervisor: Ing. Peter Chovanec, Ph.D.

Ostrava, 2022

VSB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Bachelor Thesis Assignment

Student:	**Bogdan Postolov**
Study Programme:	B2647 Information and Communication Technology
Study Branch:	2612R025 Computer Science and Technology
Title:	A Tool for Generating Data Collections, Queries and Statistical Functions Nástroj pro generování datových kolekcí, dotazů a statistické účely
The thesis language:	English

Description:

A database framework RadegastDB is a collection of persistent data structures (e.g. an array, B-tree, R-tree, Hash table and so on) implemented by a Database research group at the Department of Computer Science. Their efficiency is measured by experiments over real-world as well as generated data collections and queries. The main goal of this thesis is to implement a set of tools into the RadegastDB framework that will provide generating of collections, queries, and some statistical and support functions.

The main tasks for a student are:
1. Study testing of data structures implemented in RadegastDB.
2. Study the currently used generator of data collections and correct its shortcomings.
3. Implement the generator of various types of queries.
4. Implement a set of statistical and support functions over data collections (e.g. sorting of data or histogram).
5. Evaluate the implementation and its performance.

References:

[1] Jacob Kogan, Charles Nicholas, Marc Teboulle. Grouping Multidimensional Data. 2005
[2] Hanan Samet. Foundation of Multidimensional and Metric Data Structures. 2006

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **Ing. Peter Chovanec, Ph.D.**

Date of issue:	01.09.2021
Date of submission:	30.04.2022

doc. Ing. Petr Gajdoš, Ph.D.
Head of Department

prof. Ing. Jan Platoš, Ph.D.
Dean

Abstrakt

Hlavním cílem této práce je implementovat do frameworku RadegastDB sadu nástrojů, které zajistí generování kolekcí, dotazů a některých statistických a podpůrných funkcí. Tohoto cíle je dosaženo studiem již existujících nástrojů v rámci RadegastDB, jejich vylepšováním a implementací nových.

Klíčová slova

RadegastDB; prostorová data; datová struktura; R-strom; B–strom; n-tice; generátor; sběr dat; rozdělení; histogram; dotaz; bodový dotaz; dotaz na částečnou shodu; úzky rozsahový dotaz; rozsahový dotaz; kartézsky rozsahový dotaz; z-křivka; export a import data; SQL; DBMS;

Abstract

The main goal of this thesis is to implement a set of tools into the RadegastDB framework that will provide generating of collections, queries, and some statistical and support functions. This goal is achieved by studying the already existing tools in the RadegastDB framework, improving them, and implementing new ones.

Keywords

RadegastDB; spatial data; data structure; R-tree; B-tree; tuple; generator of data collections; distribution; histogram; query; point query; partial match query; narrow range query; range query; cartesian range query; z-order; export; SQL; DBMS;

Acknowledgement

I would like to thank all those who helped me with the work, because without them this work would not have happened. Especially to Ing. Peter Chovanec, Ph.D. for his professional guidance, patience, help and advices, without which this work wouldn't have been possible. Also, I would like to thank to my parents who have been a continuous support.

Contents

List of symbols and abbreviations

DBMS	–	DataBase Management System
DB	–	DataBase
QL	–	Query Low
QH	–	Query High
SQL	–	Structured Query Language

List of Figures

List of Tables

Chapter 1

Introduction

Implementation and improving of data structures is important part of the area of database systems. In the department of Computer Science, the database framework RadegastDB [1] is developed in last decades. This framework is implemented in *C++* and contains a set of data structures commonly used in today's relational DBMS. RadegastDB contains persistent data structures, such as B-tree, R-tree, Hash table, Linked list and Array (Section 2.1). The efficiency of this data structures is measured by the efficiency of the basic operations like inserts, updates, deletes and selects. In order to execute this operations we need some collections of data and queries. For this purpose a generator called *TupleGenerator* was implemented. However, this generator has it own flaws. Meaning that there are some issues, limitations and insufficient functions for generating the required data collections. That's why a new and improved generator will be implemented, called *CollectionGenerator* which is the work of this thesis.

The purpose of this thesis is to develop a tool for generating data collections, queries and some statistical and support functions. This is achieved in five phases, which are: Studying and testing of already existing data structures in the RadegastDB database framework (Chapter 2). Then in the chapter Current Generator (3) an overview is given of the current generator. Where its described for what is this generator used as well as its functions and disadvantages are explained. After that we'll look at the implementation of a new improved generator for data collections which is done in New Generator Generating Data Collections (Chapter 4).

Next, implementation of generator for various types of queries is done in New Generator Generating Query Collection (Chapter 5). In the fourth phase, some statistical and support functions that are implemented in the new generator are explained in Statistical and New Generator Support Functions (Chapter 6). Finally, evaluation of the implementation and its performance will be reviewed in Experiments (Chapter 7) and Conclusion (Chapter 8).

Chapter 2

Radegast DB

RadegastDB [1] is a database framework that is a collection of persistent data structures (e.g. an array [2], B-tree [3][4], R-tree [5], Hash table [2], Linked list [2]) implemented by a Database research group [1] at the Department of Computer Science.

2.1 Data Structures

Some explanation of the above mentioned data structures will be given:

An **persistent array** [2] is a set of pairs, index and value. For each index which is defined, there is a value associated with that index. In mathematical terms we call this a correspondence or a mapping. We can see an example of the persistent array data structure in the Figure 2.1.

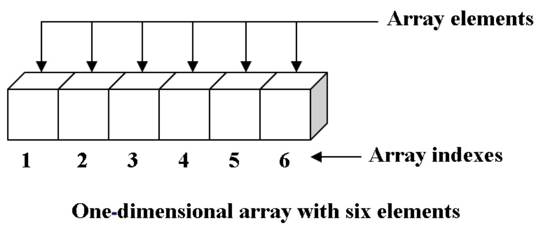


Figure 2.1: Example of Persistent Array Data Structure

A **B-tree** [3][4] is a data structure that maintains data sorted and supports logarithmic amortized searches, insertions, and deletions. It is optimized for systems that read and write big data blocks, unlike self-balancing binary search trees. It's most often found in database and file management systems. An example of the B-tree data structure can be seen in the Figure 2.2.

[1]`http://db.cs.vsb.cz`

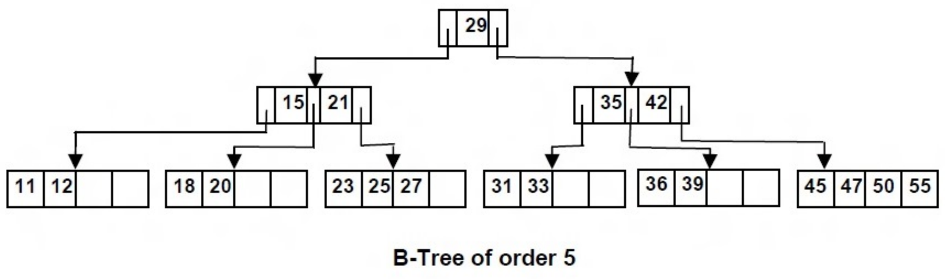
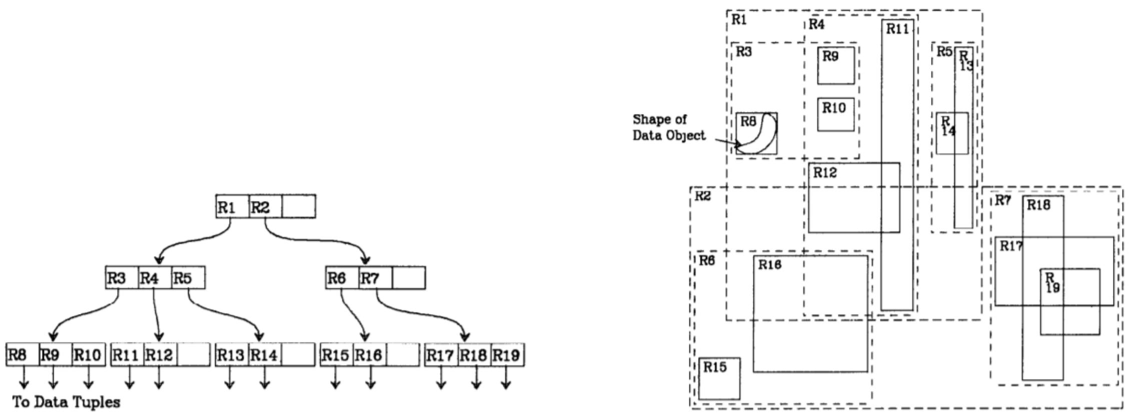


Figure 2.2: Example of the B-tree

An **R-tree** [5] is an advanced height-balanced Tree Data Structure that is widely used in production for spatial problems (like geographical map operations). An R-tree is a height-balanced tree with index records in it's leaf nodes containing pointers to data objects Nodes correspond to disk pages if the index is disk-resident, and the structure is designed so that a spatial search requires visiting only a small number of nodes. Example of the R-tree data structure can be seen in the Figure 2.3.



(a) Structure

(b) Containment and Overlapping Relationships

Figure 2.3: Example of the R-tree

2.2 Base Classes for Representation of Data

The fundamental element in all of the data structures presented is known as a tuple. The space descriptor describes an n-dimensional item called a tuple. Tuples are represented in RadegastDB by the class cTuple, and space descriptions are represented by the class cSpaceDescriptor.

2.2.1 cTuple

Represents n-dimensional homogeneous tuple for a tree data structure. It contains an array of values of the specific data type. Using this class we can work with various data types for example *int*, *unsigned int*, *float*, *double*, *char*, *unsigned char*, *wchar_t*, *short*, *unsigned short*.

1. **SetValue(unsigned int order, int value, const cDTDescriptor* pSd)**: Allows us to set a value for the appropriate data. type. For example let's say we want to set a value for an attribute of type *int*. Then we'll specify the order of the attribute into the parameter *order*, the value we want to assign will be specified into the parameter *value* and the space descriptor will be specified into the parameter *pSD*

2. **GetInt(unsigned int order, const cDTDescriptor* pSd)**: Allows us to obtain an attribute from a tuple, of the specified order. Which is specified into the parameter *order*) and the space descriptor is specified as well into the parameter *pSD*

3. **Resize(const cDTDescriptor* pSd)**: Gives us an option us to resize the tuple according to the space descriptor. Which is specified into the parameter *pSD*

4. **Clear(const cSpaceDescriptor* pSd)**: Gives us an option to set all attributes of the tuple to zero. By specifying the space descriptor into the parameter *pSD*

5. **Print(const char* string, const cSpaceDescriptor* pSd)**: Allows u to print the tuples. Where we can specify a string which will separate the tuples from one another. By specifying the desired string into the parameter *string* and the space descriptor into the parameter *pSD*

6. **SetMaxValue(unsigned int order, const cSpaceDescriptor* pSd)**: Gives us an option to set maximal value for the specified attribute of the given order. However, this function works only for *int*, *unsigned int*, and *short*. We can set a maximal value for the specified attribute by specifying the order of the attribute into the parameter *order* and the space descriptor into the parameter *pSD*

There are quite useful functions in this class however, this are some of the functions that are used in creating the new improved generator.

2.2.2 cSpaceDescriptor

cSpaceDescriptor represents the space descriptor for the cTuple class. Here all the metadata information about concrete cTuple is stored.

1. **GetDimensionType(unsigned int dim)**: Gives us an option to obtain the type of the given dimension. By specifying the dimension into the parameter *dim*

2. **SetDimensionType(unsigned int dim, cTuple *type)**: Gives us an option to set the type of the specified dimension. By specifying the dimension into the parameter *dim* and specifying the data type into the parameter *type*.

3. **GetDimensionSize(unsigned int order)**: Allows us to obtain the dimension size of the specified order. By specifying the the order into the parameter *order*.

4. **GetDimensionTypeCode(unsigned int dim)**: Allows us to obtain the code of the data type of the specified dimension. By specifying the dimension into the parameter *dim*.

In the Table 2.1 we can see the available data types and their codes.

Data Type	**Code**
cInt	*i*
cUint	*u*
cDouble	*d*
cFloat	*f*

Table 2.1: Available Data Types and Their Codes

As mentioned above this class rather support class for the class *cTuple*, so inside this class there are more support function. But we explained the most important one that are used in creating the new and improved generator.

Chapter 3

Current Generator

cTupleGenerator is a class for generating data collections. Which is used for generating data collection. This generator supports generating random tuples for a data collection as well as importing data collection from a file. We'll see some of the available functions that this generator has to offer and we'll explain what each function does.

1. **GetTuple(uint order)**: Allows us to obtain a tuple from the specified order in the parameter *order*. For example if we specify zero, it will return the first tuple in the collection

2. **GetNextRandTuple()**: Allows us to obtain a tuple from a random given order

3. **GetNextTupleFromFile()**: Allows us to obtain the next tuple from the file. Which is used when loading a data collection from a file

4. **GetNextTuple()**: Gives us an option to obtain the next tuple in the collection

5. **ResetPosition()**: Gives us an option to reset the order of the tuples. For example if the last returned tuple was the fifth tuple in the collection. Then if we call this function, the order will reset and if we call **GetNextTuple()** the first tuple will be returned

6. **GenerateRandomArray()**: Gives us an option to generate random tuples according to normal or Gaussian distribution

This generator has its own advantages and disadvantages. The advantage is that it allows us to generate random data collection and import from a file an already existing data collection. It has a very good approach for bulk loading tuples when it comes to importing a data collection. However this generator has its own flaws. The first flaw of this generator is that only generates random tuples according to the normal or Gaussian distribution. The next disadvantage of this generator is that it only generates tuple collections, there isn't a generator for query collection. Another flaw of the current generator is that there aren't much support function that will allow us to sort or shuffle the data collection. On the other hand, there is a shortness of statistical futures like generating histogram or generating SQL selects and inserts.

To summarize, the current generator *cTupleGenerator* does its job. However, as mentioned in the *Introduction*, "*implementation and improving of data structures is important part of the area of database systems*" so this was the perfect time to implement a new improved generator. Which we'll be explained and discussed in the following chapters.

Chapter 4

New Generator Generating Data Collection

In this chapter we'll talk about what is data collection and how can we generate one. Then, we'll analyze the available tuple generator, which is used to generate such data collections. Also, available distributions and futures will be discussed in this chapter.

4.1 What is Data Collection?

When we refer to "Data Collection" we mean a simple collection of data, in our case that is a collection of tuples. Not to confuse with the process "Data Collection" which has totally different meaning. In Figure 4.1 we can see an example of a data collection.

```
Dimension: 11
Tuples Count: 3

3,1,2010,2,8,442,442,421,424,205500,424
3,1,2010,2,5,442,454,422,441,194300,441
3,1,2010,2,4,455,469,439,442,233800,442
```

Figure 4.1: Data Collection Example

In mathematics, a tuple is an ordered list of elements. Related to this is an n-tuple, which in set theory is a collection (sequence) of n elements. Our data collections will be defined by *Tuples Count* and *Dimension*. *Tuples Count* refers to how many tuples will be stored or generated in the data collection. *Dimension*, defines how many elements will one tuple have. In our generator, we can either import an already existing data collection or we can generate a new random one based on our preferences, using the *Data Generator*.

4.2 Data Generator

In the previous section we defined what is data collection and what is tuple. Now we will discuss how we can use the data generator to generate a data collection or import and already existing data collection.

4.2.1 Import Data

We can import an already existing data collection from a file and load the data into the memory. In case of importing a big file into the memory, we can specify the buffer size and the size of the line. It will load part only as big as the buffer. The collection file includes number of tuples (Tuples Count) and dimension. When we load data from a file, the file is being opened using **cFileStream.h**. Once the file is opened, we look for the header where we will obtain the information of how many tuples are stored in the file and from which dimension. Next we go trough the file line by line and search "," as a delimiter for the elements of a single tuple. To know that we reach the last element of the tuple, we look for a new line. Once we obtained all the values from one tuple, we load the tuple in to the memory using a 2D array.

4.2.2 Generate Random Data

We can generate random data by given type (**cInt**, **cUInt**, **cFloat**, **cDouble**), dimension and tuples count. Also, we can be more specific and specify a minimal and a maximal value range for the elements of the tuples. We can also specify a distribution based on which random tuples will be generated. In the next subsection we'll look at the available distributions and discuss them separately.

4.3 Available Distributions

In order to generate random tuples we need to use some distribution to achieve some randomness between the tuples. The following distributions are available for use in the tuple generation: Uniform Distribution, Normal Distribution, Logonormal Distribution, Diagonal Distribution, Sierpiński Distribution and Bit Distribution. The implementation for some of the distributions mentioned above was inspired by *SpiderWeb* [6]. To test the distributions we'll visualize the generated tuples by plotting them on a graph using *SimplePlot* [7] and *Demos* [8].

4.3.1 Uniform Distribution

In probability theory and statistics, the discrete uniform distribution [9] is a symmetric probability distribution where in a finite number of values are equally likely to be observed; every one of n values has equal probability 1/n. Another way of saying "discrete uniform distribution" would be "a known, finite number of outcomes equally likely to happen". A simple example of the discrete uniform distribution is throwing a fair dice. The possible values are 1, 2, 3, 4, 5, 6, and each time the die is thrown the probability of a given score is 1/6. If two dice are thrown and their values added, the resulting distribution is no longer uniform because not all sums have equal probability.

Using this distribution, we can generate random tuples of any dimension and of any of the available types: **cInt**, **cUInt**, **cFloat**, **cDouble**.

For better understanding, we can visualize the random generated tuples by plotting them on a graph. Which can be seen in the Figure 4.2 For this purpose, we'll generate one thousand, two dimensional, random tuples using Uniform distribution and values in the range from zero to three hundred.

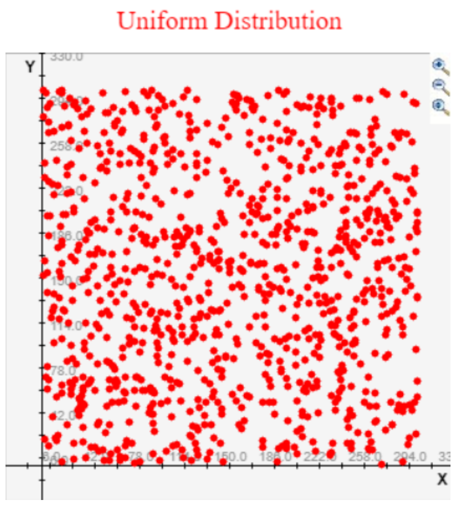


Figure 4.2: Example of Uniform Distribution

4.3.2 Normal Distribution

In statistics, a normal distribution (also known as Gaussian, Gauss, or Laplace–Gauss distribution) is a type of continuous probability distribution for a real-valued random variable [10]. The distribution can be described by two values: the mean and the standard deviation. This distribution produces random numbers around the distribution **mean** with a specific **standard deviation**. The normal distribution is a common distribution used for many kind of processes, since it is the distribution that the aggregation of a large number of independent random variables approximates to [11].

Using this distribution, we can generate random tuples of any dimension and of any of the available types: **cInt**, **cUInt**, **cFloat**, **cDouble**. In addition, we can set the *mean* by calling the function **SetMean(uint pMean)**. And we can also set the *standard deviation* by calling the function **SetStdDeviation(uint pStdDeviation)**. Both functions **SetMean(uint pMean)** and **SetStdDeviation(uint pStdDeviation)** accept parameter values greater then zero. If we don't specify the standard deviation and the mean, then they are both calculated appropriately.

We can also generate one thousand, two dimensional, random tuples within the value range, from fifty to two hundred and seventy five. Then we can see the result plotted on a graph, in the Figure 4.3.

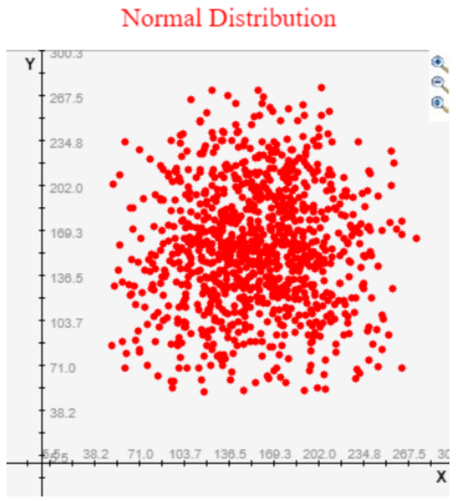


Figure 4.3: Example of Normal Distribution

4.3.3 Logonormal Distribution

In probability theory, a log-normal (or lognormal) distribution is a continuous probability distribution of a random variable whose logarithm is normally distributed. Thus, if the random variable X is log-normally distributed, then $Y = ln(X)$ has a normal distribution. Equivalently, if Y has a normal distribution, then the exponential function of Y, $X = exp(Y)$, has a log-normal distribution. A random variable which is log-normally distributed takes only positive real values. The distribution is occasionally referred to as the Galton distribution or Galton's distribution, after Francis Galton [12].

Using this distribution, we can generate random tuples of any dimension and of any of the available types: **cInt**, **cUInt**, **cFloat**, **cDouble**. In this function the mean and the standard deviation are calculated appropriately depending on the number of tuples and their values.

We can visualize the random generated tuples by plotting them on a graph. For this purpose we'll generate one thousand, two dimensional random tuples with values in range from zero to three hundred and fifty seven. The result can be seen in the Figure 4.4.

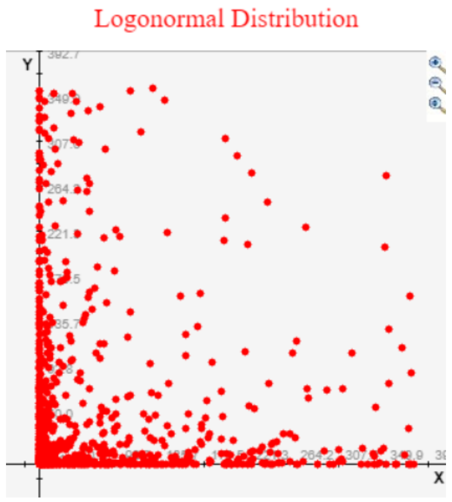


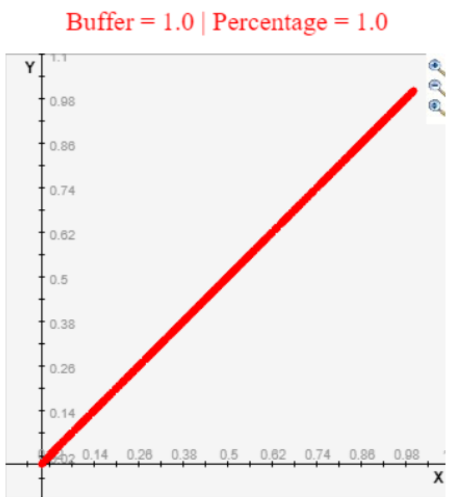
Figure 4.4: Example of Logonormal Distribution

4.3.4 Diagonal Distribution

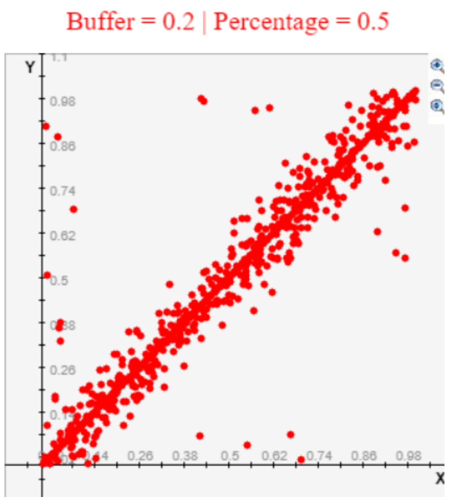
Using Diagonal distribution [13], we can generate two dimensional random tuples with real values from one of the available data types: **cDouble** or **cFloat**. The result of the random generated tuples always represent a diagonal line. The thicknes of the line depends on the number of tuples. As well of the percentage (ratio) of the points that are exactly on the line, the *percentage* can be set by calling the function **SetPercentage(double pPercentage)**. Also, it depends on the size of the buffer around the line where additional points are scattered, the *buffer* can be set by calling the function **SetBuffer(double pBuffer)**. Both functions **SetPercentage(double pPercentage)** and **SetBuffer(double pBuffer)** accept parameters values in range from zero to one.

For better understanding we'll generate one thousand, two dimensional, random tuples and we'll visualize them by plotting the random generated tuples. And we'll look at four cases in the Figure 4.5. In the first case, that is Figure 4.5a, we'll set the *buffer* 1.0 and the *percentage* at 1.0 as well. Then the second case, in Figure 4.5b, we'll set the *buffer* at 0.2 and the *percentage* at 0.5. In the third case, which is presented in Figure 4.5c, we'll set the *buffer* at 0.5 and the *percentage* at 0.7.

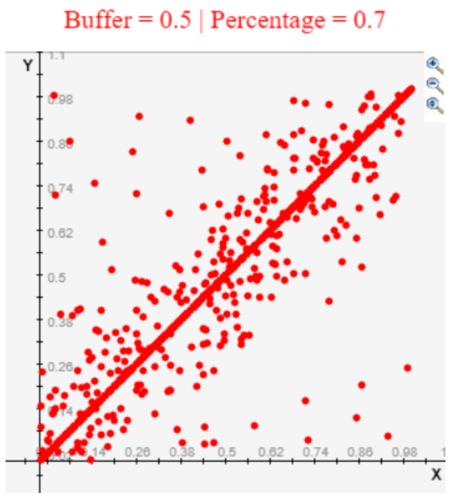
Finally, in the fourth final case, in Figure 4.5d, we'll set the *buffer* at 1.0 and the *percentage* at 0.2.



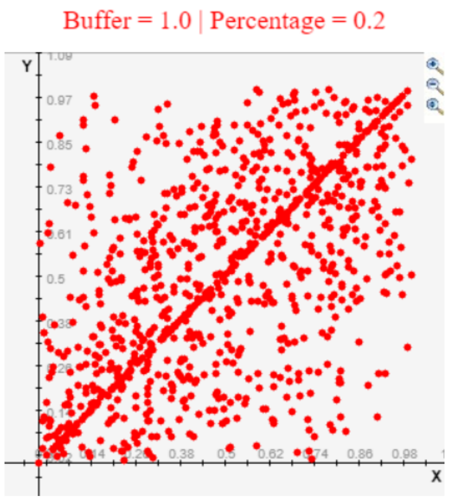
(a) *buffer* = 1.0 and *percentage* = 1.0



(b) *buffer* = 0.2 and *percentage* = 0.5



(c) *buffer* = 0.5 and *percentage* = 0.7



(d) *buffer* = 1.0 and *percentage* = 0.2

Figure 4.5: Example of Diagonal Distribution

4.3.5 Sierpiński Distribution

This distribution is based on the Sierpiński triangle [14][13]. The Sierpiński triangle (sometimes spelled Sierpinski), also called the Sierpiński gasket or Sierpiński sieve, is a fractal attractive fixed set with the overall shape of an equilateral triangle, subdivided recursively into smaller equilateral triangles .

Using this distribution, we can generate random tuples of any dimension and of any of the available types: **cFloat**, **cDouble**. The first three points are the corners of the triangle. The successive points are generated starting from the current point and computing the middle point between the current point and one of the vertices of the triangle chosen according to the random function **Dice(int n)**.

We can visualize the Sierpiński distribution by generating one thousand, two dimensional, random tuples and plot them on a graph. The result can be seen in the Figure 4.6.

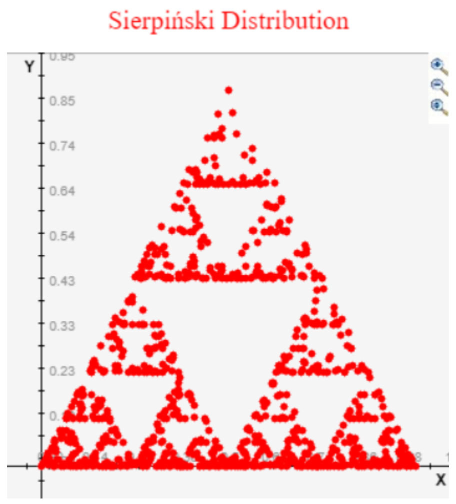


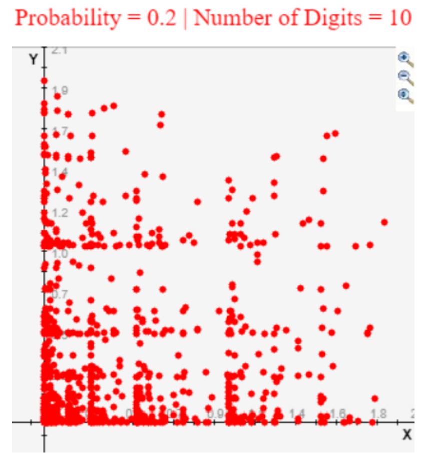
Figure 4.6: Example of Sierpiński Distribution

4.3.6 Bit Distribution

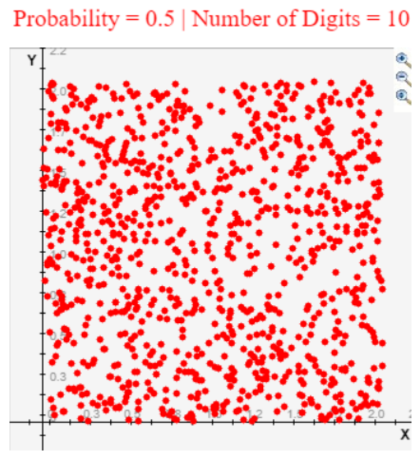
Bit distribution [13] generates skewed tuples by introducing a rule for generating the coordinates of the points by assigning higher probability to a subset of coordinates. For instance in the Bit distribution, the point coordinates are generated as a bit string of a fixed length where each bit is set with a fixed probability in the range from zero to one.

Using this distribution, we can generate random tuples of any dimension and of any of the available types: **cFloat**, **cDouble**. For this distribution we can set the *probability* by calling the function **SetProbability(double pProbability)**. Where the *probability* represents a fixed probability of setting each bit independently to one. Also, we can set the *number of digits* by calling the function **SetDigits(int pDigits)**. Where *number of digits* represents the number of binary digits after the fraction point.

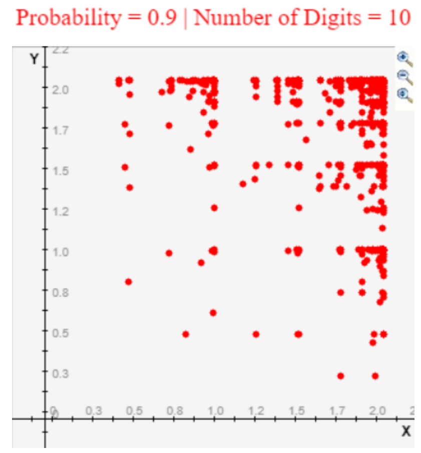
We can visualize this distribution by generating one thousand, two dimensional, random tuples and plot them on a graph. We'll look at four cases in the Figure 4.7. In the first two cases, in the Figure the *number of digits* will be set to ten. In addition, in the first case presented in Figure 4.7a, we'll set the *probability* at 0.2. Next, in the second case in Figure 4.7b, we'll set the *probability* at 0.5. Last but not least in the third case presented in Figure 4.7c, we'll set *probability* at 0.9. Finally, in the last case in Figure 4.7d, we'll set the *number of digits* to one.



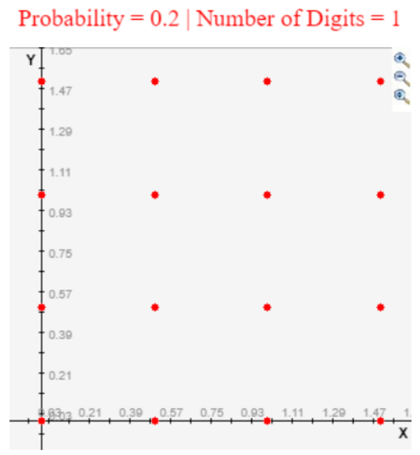
(a) *probability* = 0.2 and *number of digits* = 10



(b) *probability* = 0.5 and *number of digits* = 10



(c) *probability* = 0.9 and *number of digits* = 10



(d) *probability* = 0.2 and *number of digits* = 1

Figure 4.7: Example of Bit Distribution

Chapter 5

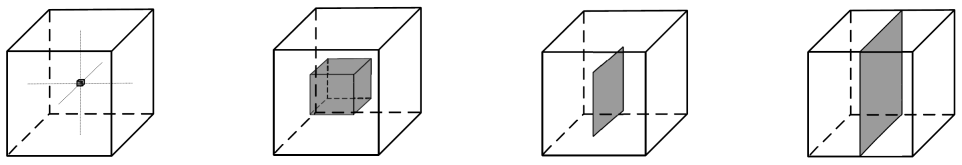
New Generator Generating Query Collection

In this chapter we'll talk about what are queries and how can we generate them. Then, we'll analyze the available types of query generators. The new generator provides generators for following types of queries: Point Queries, Partial Match Queries, Narrow Range Queries, Range Queries and Cartesian Range Queries.

5.1 What is Query?

In databases a *query* is a request for data or information from a database table or combination of tables. This data may be generated as results returned by Structured Query Language (SQL) or as pictorials, graphs or complex results, e.g., trend analyses from data-mining tools [15].

Various types of queries, such as range queries and other similarity queries, are supported by multidimensional access methods. The range query returns all tuples from a multidimensional space contained within a query rectangle defined by an interval or a point in each dimension (attribute). In the Figure 5.1 we can see an an example of a point query, range query, partial match query, narrow range query.



(a) Point Query (b) Range Query (c) Narrow Range Query (d) Partial Match Query

Figure 5.1: Example of Different Types Of Queries

The lower boundary is called *QL* in short of **Query Low**. On the other hand, the upper boundary is called *QH* in short of **Query High**. The attribute values from *QL* always must be smaller or equal to the attribute values from *QH*. This two boundaries form query window or also known as query box, which can be seen in the Figure 5.2. Where *R1* represents the query window and the points *P1*, *P2*, and *P3* are inside the query window. While the points *P4* and *P5* are outside the query window.

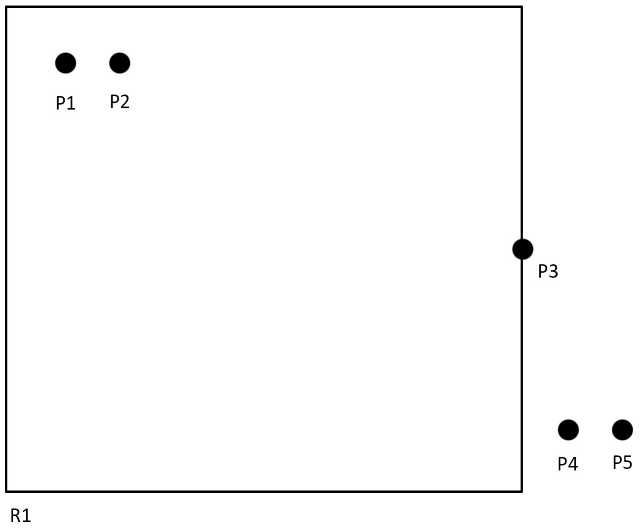


Figure 5.2: Example of Query Window

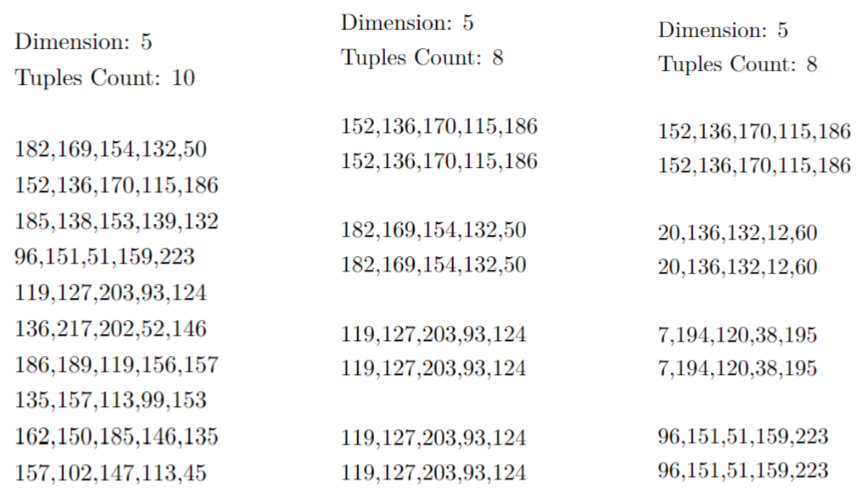
5.2 Point Queries

A point query (also known as an exact match query) limits all attributes to a point. Such queries can be generated by calling the function **GeneratePointQueries(uint pQueriesCount, double pPercentage**. This function takes the following parameters:

- **uint pQueriesCount**: number of queries to be generated
- **double pPercentage**: percentage of *empty queries*

The function takes two random tuples and generates one point query. In case we have specified that we want a certain *percentage* of *empty queries* in the collection. Empty queries represent data that is not part of the data collection. For this purpose certain number of empty queries are generated and they are verified that they do not return any result. If they don't return any result then they are added to the query collection.

To test this function, we can generate four point queries from previously generated ten, five dimensional, random tuples of type *cUInt* with values in range from zero to two hundred and fifty, generated randomly by using normal distribution (See Figure 5.3a). We'll see two cases in the Figure 5.3. In the first case, presented in Figure 5.3b, we'll set the *percentage* of empty queries to 0. And in the second case, in Figure 5.3c, we'll set the *percentage* of empty queries to 0.5.



```
Dimension: 5
Tuples Count: 10

182,169,154,132,50
152,136,170,115,186
185,138,153,139,132
96,151,51,159,223
119,127,203,93,124
136,217,202,52,146
186,189,119,156,157
135,157,113,99,153
162,150,185,146,135
157,102,147,113,45
```

(a) Tuples

```
Dimension: 5
Tuples Count: 8

152,136,170,115,186
152,136,170,115,186

182,169,154,132,50
182,169,154,132,50

119,127,203,93,124
119,127,203,93,124

119,127,203,93,124
119,127,203,93,124
```

(b) Percentage=0.0

```
Dimension: 5
Tuples Count: 8

152,136,170,115,186
152,136,170,115,186

20,136,132,12,60
20,136,132,12,60

7,194,120,38,195
7,194,120,38,195

96,151,51,159,223
96,151,51,159,223
```

(c) Percentage=0.5

Figure 5.3: Example of Two Point Query Collections Generated for Data Collection

5.3 Partial Match Queries

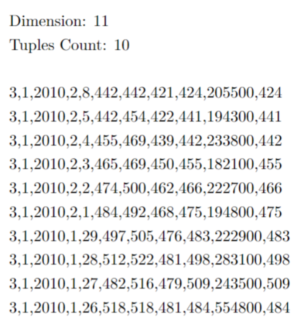
A partial match query limits some attributes to a point while leaving others unspecified. In this generator, we can specify the *percentage* of *empty queries* we want to be generated in the collection. This types of queries can be generated by calling the function **GeneratePartialMatchQueries(uint pQueriesCount, double pPercentage)**. This function takes the following parameters:

- **uint pQueriesCount**: number of queries to be generated

- **double pPercentage**: percentage of *empty queries*

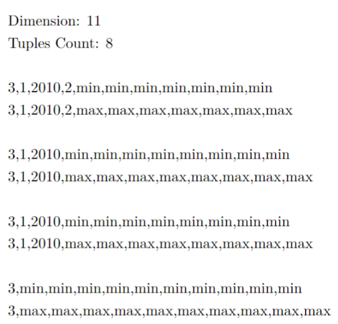
This generator is similar to the point query generator. This function takes two random tuples, compares each element of the two random tuples and generates one partial match query. First it checks the first elements, if they are same then it moves to the second one and so on... If two different elements occur then for the element of the first tuple, the value *min* is set. For the element of the second tuple, the value *max* is set. In case we have specified that we want a certain percentage of *empty queries* in the collection. Then a certain number of empty queries are generated and they are verified that they do not return any result. If they don't return any result then the same check mentioned above is performed and they are added to the query collection.

To test this function, we can generate four partial match queries from an already imported tuples collection of type *cUInt*. There are ten, eleven dimensional tuples in the tuples collection (See Figure 5.4a). We'll see two cases in the Figure 5.4. In the first case, presented in Figure 5.4b, we'll set the *percentage* of empty queries to 0. And in the second case, in Figure 5.4c, we'll set the *percentage* of empty queries to 0.5.



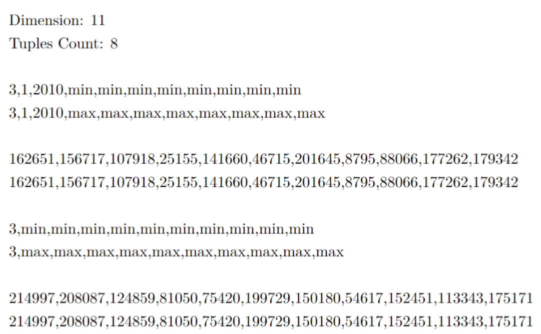
```
Dimension: 11
Tuples Count: 10

3,1,2010,2,8,442,442,421,424,205500,424
3,1,2010,2,5,442,454,422,441,194300,441
3,1,2010,2,4,455,469,439,442,233800,442
3,1,2010,2,3,465,469,450,455,182100,455
3,1,2010,2,2,474,500,462,466,222700,466
3,1,2010,2,1,484,492,468,475,194800,475
3,1,2010,1,29,497,505,476,483,222900,483
3,1,2010,1,28,512,522,481,498,283100,498
3,1,2010,1,27,482,516,479,509,243500,509
3,1,2010,1,26,518,518,481,484,554800,484
```

(a) Tuples



```
Dimension: 11
Tuples Count: 8

3,1,2010,2,min,min,min,min,min,min,min
3,1,2010,2,max,max,max,max,max,max,max

3,1,2010,min,min,min,min,min,min,min,min
3,1,2010,max,max,max,max,max,max,max,max

3,1,2010,min,min,min,min,min,min,min,min
3,1,2010,max,max,max,max,max,max,max,max

3,min,min,min,min,min,min,min,min,min,min
3,max,max,max,max,max,max,max,max,max,max
```

(b) Percentage = 0.0



```
Dimension: 11
Tuples Count: 8

3,1,2010,min,min,min,min,min,min,min,min
3,1,2010,max,max,max,max,max,max,max,max

162651,156717,107918,25155,141660,46715,201645,8795,88066,177262,179342
162651,156717,107918,25155,141660,46715,201645,8795,88066,177262,179342

3,min,min,min,min,min,min,min,min,min,min
3,max,max,max,max,max,max,max,max,max,max

214997,208087,124859,81050,75420,199729,150180,54617,152451,113343,175171
214997,208087,124859,81050,75420,199729,150180,54617,152451,113343,175171
```

(c) Percentage = 0.5

Figure 5.4: Example of Two Partial Match Query Collections Generated for Data Collection

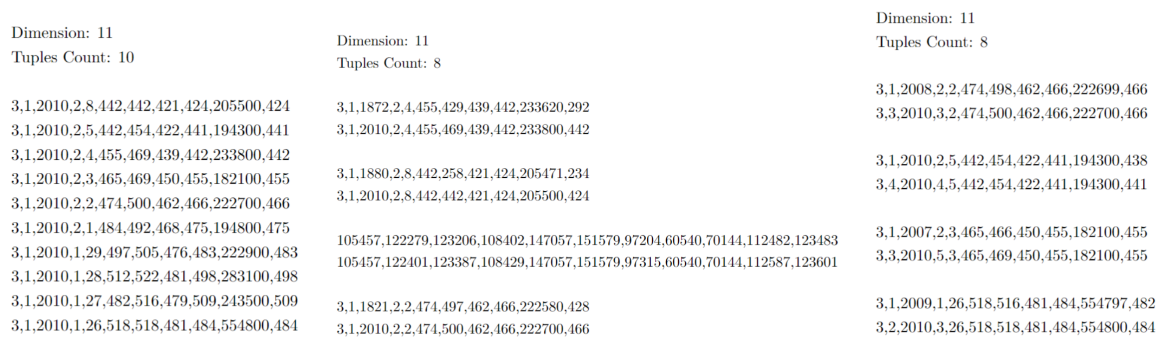
5.4 Narrow Range Queries

A narrow range query is one that restricts at least one attribute to a narrow interval. In this generator, we can specify a value for the *dispersion* [16], *boolean mask* (0 = interval attribute, 1 = point attribute), and the *percentage* of *empty queries* we want to be generated in the collection. This types of queries can be generated by calling the function **GenerateNarrowRangeQueries(uint pQueriesCount, uint pDispersion, char* pMask, double pPercentage)**. This function takes the following parameters:

- **uint pQueriesCount**: number of queries to be generated
- **uint pDispersion**: value for the *dispersion*, must be greater or equal than one
- **char* pMask**: string of a boolean mask; 0 = interval attribute, 1 = point attribute. E.g. "1,0,0,1,1,0,1,1"
- **double pPercentage**: percentage of *empty queries*

This generator works similarly as the point query generator (Section 5.2). When the function is called, firstly it checks if we have specified any boolean mask. If we haven't specified any boolean mask then a random one is generated by calling the function **GenerateMask(uint pDimCount)** where the dimension is specified into the parameter *pDimCount.* In case we have specified a *boolean mask*, then it is converted into a boolean array using the function **ConvertToBoolArray(bool* pArray, char* pString)** where we pass the boolean array into the parameter *pArray* and the string of the boolean mask that needs to be converted into boolean values. Next, the element of the *QL* is checked if it's larger than the specified *dispersion.* If the element is greater than the specified *dispersion*, then a random number is generated in the range from zero to the specified *dispersion*, using uniform distribution. Then if the *boolean mask* returns *1* for the given element, the unmodified value from the tuple is set for the given element of the *QL.* If the *boolean mask* returns *0* then the value for the element of the *QL* is modified by $tuple - (random(dispersion))$. *random(dispersion)* is the random number that is mentioned above. The similar process is done for the elements of the *QH*, the difference is in the check for the random number generation. It checks if the sum of the value of the given element and the value of the specified *dispersion*, is smaller than the *domain* which is set to one hundred. Another difference is if the *boolean mask* returns zero for the given tuple element, then the value is modified by $tuple + (random(dispersion))$. Finally, In case we have specified that we want a certain percentage of *empty queries* in the collection. Then a certain number of empty queries are generated and they are verified that they do not return any result. If they don't return any result then the check mentioned above is performed and they are added to the query collection.

To test this function, we can generate four narrow range queries from an already imported tuples collection of type *cUInt*. There are ten, eleven dimensional tuples in the tuples collection (See Figure 5.5a). We'll see two cases in the Figure 5.5. In the first case, presented in Figure 5.5b, we'll set the *dispersion* to two hundred, *boolean mask* to *"1,0,0,0,1,1,0,1,1,0,0"* and the *percentage* of empty queries to 0.2. Next, in the second case, in Figure 5.5c, we'll set the *dispersion* to three and the *percentage* of empty queries to 0.



```
Dimension: 11
Tuples Count: 10

3,1,2010,2,8,442,442,421,424,205500,424
3,1,2010,2,5,442,454,422,441,194300,441
3,1,2010,2,4,455,469,439,442,233800,442
3,1,2010,2,3,465,469,450,455,182100,455
3,1,2010,2,2,474,500,462,466,222700,466
3,1,2010,2,1,484,492,468,475,194800,475
3,1,2010,1,29,497,505,476,483,222900,483
3,1,2010,1,28,512,522,481,498,283100,498
3,1,2010,1,27,482,516,479,509,243500,509
3,1,2010,1,26,518,518,481,484,554800,484
```

(a) Tuples

```
Dimension: 11
Tuples Count: 8

3,1,1872,2,4,455,429,439,442,233620,292
3,1,2010,2,4,455,469,439,442,233800,442

3,1,1880,2,8,442,258,421,424,205471,234
3,1,2010,2,8,442,442,421,424,205500,424

105457,122279,123206,108402,147057,151579,97204,60540,70144,112482,123483
105457,122401,123387,108429,147057,151579,97315,60540,70144,112587,123601

3,1,1821,2,2,474,497,462,466,222580,428
3,1,2010,2,2,474,500,462,466,222700,466
```

(b) First Collection

```
Dimension: 11
Tuples Count: 8

3,1,2008,2,2,474,498,462,466,222699,466
3,3,2010,3,2,474,500,462,466,222700,466

3,1,2010,2,5,442,454,422,441,194300,438
3,4,2010,4,5,442,454,422,441,194300,441

3,1,2007,2,3,465,466,450,455,182100,455
3,3,2010,5,3,465,469,450,455,182100,455

3,1,2009,1,26,518,516,481,484,554797,482
3,2,2010,3,26,518,518,481,484,554800,484
```

(c) Second Collection

Figure 5.5: Example of Two Query Collections Generated for Data Collection

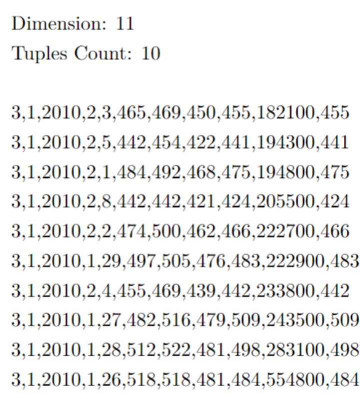
5.5 Range Queries

Range query[17] is general query where each attribute is defined by query window. The size of the query will be defined by query window. It means that we can specify how many tuples will one query contain. For this purpose we need to somehow sort the tuples in the collection. This is why the function Z-Order is being implemented (See Section 6.9). This way we can represent multidimensional tuples into single dimensional *Z addresses*. To generate *range queries*, we need to call the function **GenerateRangeQueries(uint pQueriesCount, uint pQuerysetSizeStart, uint pQuerysetSizeEnd)**. Using this function we can generate *range queries* of the following type: **cInt** and **cUInt**. This function takes the following parameters:

- **uint pQueriesCount**: number of queries to be generated
- **uint pQuerysetSizeStart**: minimal number of tuples contained in one query
- **uint pQuerysetSizeEnd**: maximal number of tuples contained in one query

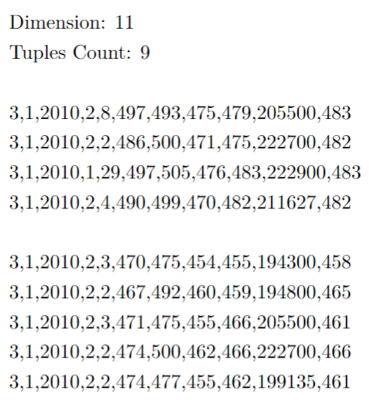
This generator works in the following way: Firstly it generates *Z addresses* for all the tuples in the collection. Then we sort the tuples by their *Z addresses*. Next, it generates random tuple indexes for QL and QH, and it checks if there are enough tuples in the generated query window. The next step is to obtain the tuples inside the given query window and to check for each element of the tuple if it is between the QL and QH tuples. The value of the tuple element must not be smaller than the element of the QL and must not be greater than the element of the QH. If an element isn't between the QL and QH tuples, then the value of the element is modified, so the tuple can fit inside the query window. Then the function continues to check if the relevant tuples fit inside the query window and also keeps track of the generated tuples inside the query, not to exceed the maximal number of tuples inside one query.

To test this function, we can generate two range queries from an already imported tuples collection of type *cInt* (See Figure 5.6a). We'll set each query to contain between two to eight tuples. The generated *range queries* can be seen in the figure 5.6b.



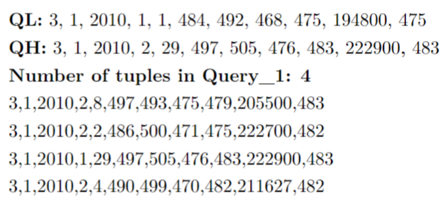
```
Dimension: 11
Tuples Count: 10

3,1,2010,2,3,465,469,450,455,182100,455
3,1,2010,2,5,442,454,422,441,194300,441
3,1,2010,2,1,484,492,468,475,194800,475
3,1,2010,2,8,442,442,421,424,205500,424
3,1,2010,2,2,474,500,462,466,222700,466
3,1,2010,1,29,497,505,476,483,222900,483
3,1,2010,2,4,455,469,439,442,233800,442
3,1,2010,1,27,482,516,479,509,243500,509
3,1,2010,1,28,512,522,481,498,283100,498
3,1,2010,1,26,518,518,481,484,554800,484
```

(a) Generated Tuples



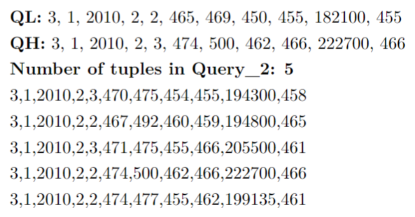
```
Dimension: 11
Tuples Count: 9

3,1,2010,2,8,497,493,475,479,205500,483
3,1,2010,2,2,486,500,471,475,222700,482
3,1,2010,1,29,497,505,476,483,222900,483
3,1,2010,2,4,490,499,470,482,211627,482

3,1,2010,2,3,470,475,454,455,194300,458
3,1,2010,2,2,467,492,460,459,194800,465
3,1,2010,2,3,471,475,455,466,205500,461
3,1,2010,2,2,474,500,462,466,222700,466
3,1,2010,2,2,474,477,455,462,199135,461
```

(b) Range Query Collection



```
QL: 3, 1, 2010, 1, 1, 484, 492, 468, 475, 194800, 475
QH: 3, 1, 2010, 2, 29, 497, 505, 476, 483, 222900, 483
Number of tuples in Query_1: 4
3,1,2010,2,8,497,493,475,479,205500,483
3,1,2010,2,2,486,500,471,475,222700,482
3,1,2010,1,29,497,505,476,483,222900,483
3,1,2010,2,4,490,499,470,482,211627,482
```

(c) First Range Query



```
QL: 3, 1, 2010, 2, 2, 465, 469, 450, 455, 182100, 455
QH: 3, 1, 2010, 2, 3, 474, 500, 462, 466, 222700, 466
Number of tuples in Query_2: 5
3,1,2010,2,3,470,475,454,455,194300,458
3,1,2010,2,2,467,492,460,459,194800,465
3,1,2010,2,3,471,475,455,466,205500,461
3,1,2010,2,2,474,500,462,466,222700,466
3,1,2010,2,2,474,477,455,462,199135,461
```

(d) Second Range Query

Figure 5.6: Example of a Range Query Collections Generated for Data Collection

In the Figure 5.6c we can see the first generated range query and its query window. Next, in the Figure 5.6d we can see the second generated range query and its query window. Finally, in the Figure 5.6b we can see the range query collection generated.

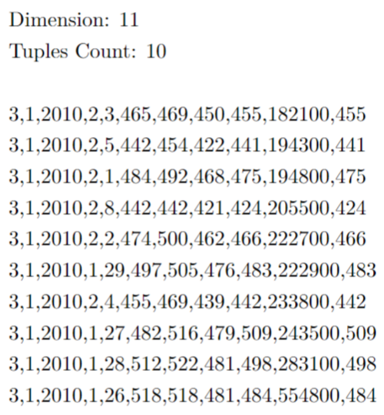
5.6 Cartesian Range Queries

Cartesian range query is query where each attribute can be defined by a set of query windows. In this generator, we will care about the size of the result as well as the maximal number of query windows for each query window. By using this generator we can generate cartesian range queries of the following type: **cInt** and **cUInt**. In order to generate cartesian range queries we need to call the function **GenerateCartesianRangeQueries(uint pQueriesCount, uint pQuerysetSizeStart, uint pQuerysetSizeEnd, uint pIntervals)**. This function takes the following parameters:

- **uint pQueriesCount**: number of queries to be generated
- **uint pQuerysetSizeStart**: minimal number of tuples contained in one query
- **uint pQuerysetSizeEnd**: maximal number of tuples contained in one query
- **uint pIntervals**: number of query windows

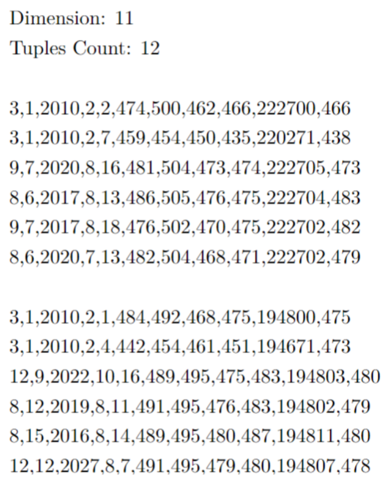
This generator works in the following way. The generator will generate range queries for each cartesian range query. First, it sets the minimal and maximal number of tuples to be generated for the range queries. The minimal number of tuples in one range query is set by: *minimal number of tuples in one query / number of query windows.* Then the maximal number of tuples in one range query is set by: *maximal number of tuples in one query / number of query windows.* Next it will check if the generated range queries are in the given query windows and will merge them together in the cartesian range query collection.

To test this function, we can generate two cartesian range queries from an already imported tuples collection of type *cInt* (See Figure 5.7a). We'll set each query to contain between two to eight tuples and defined by two query windows. We can see the two generated query windows (First Interval and Second Interval labeled in the Figure) and its data for the first generated cartesian range query in the Figure 5.7c. Next, in the Figure 5.7d we can see the two generated query windows (First Interval and Second Interval labeled in the Figure) and its data for the second cartesian range query and the second generated cartesian range query. Finally, in the Figure 5.7b we can see the whole cartesian range query collection generated.



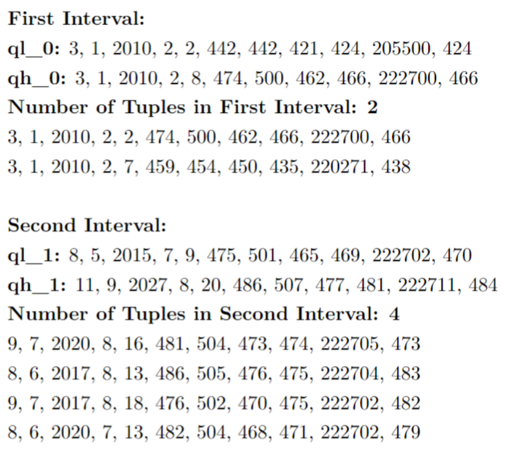
```
Dimension: 11
Tuples Count: 10

3,1,2010,2,3,465,469,450,455,182100,455
3,1,2010,2,5,442,454,422,441,194300,441
3,1,2010,2,1,484,492,468,475,194800,475
3,1,2010,2,8,442,442,421,424,205500,424
3,1,2010,2,2,474,500,462,466,222700,466
3,1,2010,1,29,497,505,476,483,222900,483
3,1,2010,2,4,455,469,439,442,233800,442
3,1,2010,1,27,482,516,479,509,243500,509
3,1,2010,1,28,512,522,481,498,283100,498
3,1,2010,1,26,518,518,481,484,554800,484
```

(a) Generated Tuples



```
Dimension: 11
Tuples Count: 12

3,1,2010,2,2,474,500,462,466,222700,466
3,1,2010,2,7,459,454,450,435,220271,438
9,7,2020,8,16,481,504,473,474,222705,473
8,6,2017,8,13,486,505,476,475,222704,483
9,7,2017,8,18,476,502,470,475,222702,482
8,6,2020,7,13,482,504,468,471,222702,479

3,1,2010,2,1,484,492,468,475,194800,475
3,1,2010,2,4,442,454,461,451,194671,473
12,9,2022,10,16,489,495,475,483,194803,480
8,12,2019,8,11,491,495,476,483,194802,479
8,15,2016,8,14,489,495,480,487,194811,480
12,12,2027,8,7,491,495,479,480,194807,478
```

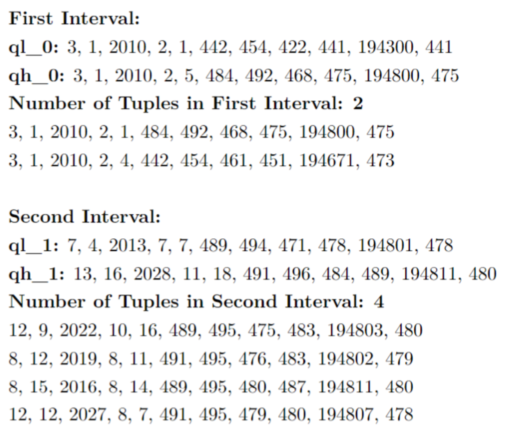
(b) Generated Queries



First Interval:
ql_0: 3, 1, 2010, 2, 2, 442, 442, 421, 424, 205500, 424
qh_0: 3, 1, 2010, 2, 8, 474, 500, 462, 466, 222700, 466
Number of Tuples in First Interval: 2
3, 1, 2010, 2, 2, 474, 500, 462, 466, 222700, 466
3, 1, 2010, 2, 7, 459, 454, 450, 435, 220271, 438

Second Interval:
ql_1: 8, 5, 2015, 7, 9, 475, 501, 465, 469, 222702, 470
qh_1: 11, 9, 2027, 8, 20, 486, 507, 477, 481, 222711, 484
Number of Tuples in Second Interval: 4
9, 7, 2020, 8, 16, 481, 504, 473, 474, 222705, 473
8, 6, 2017, 8, 13, 486, 505, 476, 475, 222704, 483
9, 7, 2017, 8, 18, 476, 502, 470, 475, 222702, 482
8, 6, 2020, 7, 13, 482, 504, 468, 471, 222702, 479

(c) First Cartesian Range Query



First Interval:
ql_0: 3, 1, 2010, 2, 1, 442, 454, 422, 441, 194300, 441
qh_0: 3, 1, 2010, 2, 5, 484, 492, 468, 475, 194800, 475
Number of Tuples in First Interval: 2
3, 1, 2010, 2, 1, 484, 492, 468, 475, 194800, 475
3, 1, 2010, 2, 4, 442, 454, 461, 451, 194671, 473

Second Interval:
ql_1: 7, 4, 2013, 7, 7, 489, 494, 471, 478, 194801, 478
qh_1: 13, 16, 2028, 11, 18, 491, 496, 484, 489, 194811, 480
Number of Tuples in Second Interval: 4
12, 9, 2022, 10, 16, 489, 495, 475, 483, 194803, 480
8, 12, 2019, 8, 11, 491, 495, 476, 483, 194802, 479
8, 15, 2016, 8, 14, 489, 495, 480, 487, 194811, 480
12, 12, 2027, 8, 7, 491, 495, 479, 480, 194807, 478

(d) Second Cartesian Range Queries

Figure 5.7: Example of Cartesian Range Query Collection Generated for Data Collection

Chapter 6

New Generator Statistical and Support Functions

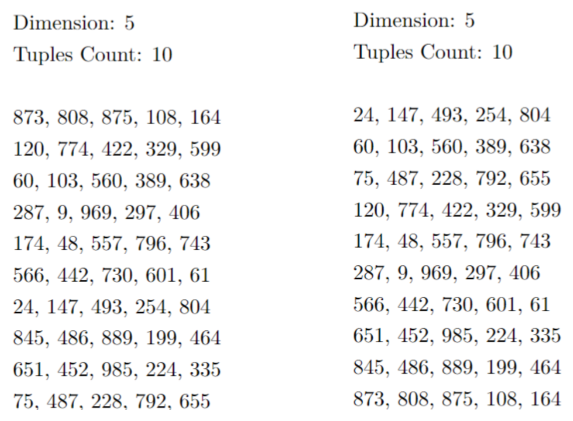
In this chapter we'll take a look at the statistical and support functions, implemented in the new generator, namely functions for sorting and shuffling of data/query collections, exporting of data/-query collection, generators of sql insert/select commands, generators of histograms and so on.

6.1 Sort Data Collection

Its useful to have a public support function that will allow us to sort the tuples alphabetically. For this purpose the function **SortTuples(TypeOfSort pTypeOfSortl)** is implemented, which allows us to sort a collection of tuples. This function takes the following parameters:

- **TypeOfSort pTypeOfSort**: type of sort that will be performed: alphabetical or Z-Order Sort

This function works in the following way: First, it checks what type of sort to be performed, if we have specified that we want *alphabetical* sort. Then the tuples are sorted alphabetically. Bubble sort algorithm is used for this purpose. We can see an example of this function in Figure 6.1. Where we will sort a collection of ten, five dimensional, random generated tuples.

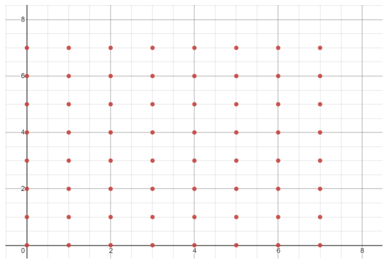


```
Dimension: 5                      Dimension: 5
Tuples Count: 10                  Tuples Count: 10

873, 808, 875, 108, 164           24, 147, 493, 254, 804
120, 774, 422, 329, 599           60, 103, 560, 389, 638
60, 103, 560, 389, 638            75, 487, 228, 792, 655
287, 9, 969, 297, 406             120, 774, 422, 329, 599
174, 48, 557, 796, 743            174, 48, 557, 796, 743
566, 442, 730, 601, 61            287, 9, 969, 297, 406
24, 147, 493, 254, 804            566, 442, 730, 601, 61
845, 486, 889, 199, 464           651, 452, 985, 224, 335
651, 452, 985, 224, 335           845, 486, 889, 199, 464
75, 487, 228, 792, 655            873, 808, 875, 108, 164
```

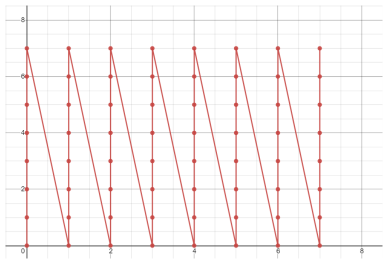
(a) Generated Tuples (b) Sorted Tuples

Figure 6.1: Example of Alphabetical Sort

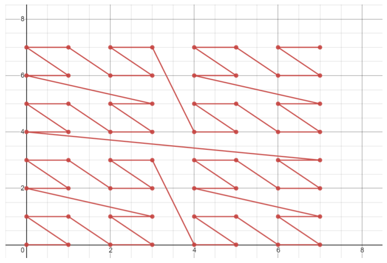
Also, we can see an example of the two available sort options in Figure 6.2 performed on sixty four, two dimensional, tuples with value within range from zero to seven. For this purpose, we'll plot the generated and sorted tuples using *Demos* [8]. In Figure 6.2a we can see the generated tuples. Then we can see an example of *alphabetical sort* performed in Figure 6.2b and in 6.2c an example of *Z-Order sort* performed.



(a) Generated Tuples



(b) Alphabetical Sort



(c) Z-Order Sort

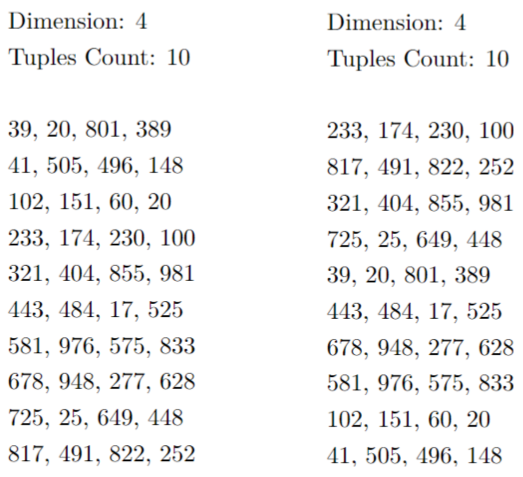
Figure 6.2: Example of Different Types of Sort

6.2 Shuffle Data Collection

On the other hand, another useful support public function that is implemented in **cCollection-Generator** is the option to unsort or shuffle the tuples, the function is called **UnsortTuples(uint pNumberOfSwaps)** and it takes one optional parameter **pNumberOfSwaps**. This function has the following parameters:

- **uint pNofShuffles**: number of swaps or shuffles

To shuffle the tuples, the function takes two random tuples and switches their places. We repeat this process for the specified number of swaps (pNumberOfSwaps). If no value is assigned for the optional parameter than a default value of ten is assigned. We can test this function (See Figure 6.3) by generating ten, four dimensional, random tuples and first we'll use the function **SortTuples()** to sort them that can be seen in the Figure 6.3a. Then we'll shuffle them using the function **UnsortTuples()**. The result can be seen in Figure 6.3b.



```
Dimension: 4              Dimension: 4
Tuples Count: 10          Tuples Count: 10

39, 20, 801, 389          233, 174, 230, 100
41, 505, 496, 148         817, 491, 822, 252
102, 151, 60, 20          321, 404, 855, 981
233, 174, 230, 100        725, 25, 649, 448
321, 404, 855, 981        39, 20, 801, 389
443, 484, 17, 525         443, 484, 17, 525
581, 976, 575, 833        678, 948, 277, 628
678, 948, 277, 628        581, 976, 575, 833
725, 25, 649, 448         102, 151, 60, 20
817, 491, 822, 252        41, 505, 496, 148
```

(a) Sorted Tuples (b) Shuffled Tuples

Figure 6.3: Example of Shuffled Tuples

6.3 Sort Query Collection

Also, there is a private function called **SortQueries(cTuple** pCollecetion, uint pStart, uint pEnd)** which sorts the queries in ascending order. Which is similar to the function **SortTuples()** (Section 6.1). This function takes the following parameters:

- **cTuple** pCollection**: array of a collection that will be sorted
- **uint pStart**: from where to start the sorting, from which tuple index
- **uint pEnd**: to which tuple index the sorting will be done

Again, bubble sort algorithm is used for sorting the queries. This function is a support function and is set as private. Since sorting and shuffling is done upon the generation of queries. This function is used upon generating various types of queries.

6.4 Shuffle Query Collection

Identical to **UnsortTuples()** (Section 6.2) is the private function **ShuffleQueries(cTuple** pCollection, uint pDimension, uint pQueryCount, uint pNofShuffles)** which shuffles the queries randomly. This function has the following parameters:

- **cTuple** pCollection**: array of a collection that will be shuffled
- **uint pDimension**: dimension of the tuples
- **uint pQueryCount**: number of tuples inside the collection
- **uint pNofShuffles**: number of swaps or shuffles

The function takes two random queries and switches their places. The process is repeated ten times if no value is given for **pNofshuffles**. This function is used upon generating various types of queries.

6.5 Export Collection to a File

This statistical function allows us to export generated tuples or queries into a file. This function is called **ExportCollection(CollectionToExport pCollection, char pCollectionName*)** and has the following parameters:

- **CollectionToExport pCollection**: what type of collection will be exported, tuples or queries

- **char pCollectionName***: name of file, the default value for this optional parameter is **ExportedCoillection**. This name will be set for the exported collection if we don't specify any file name.

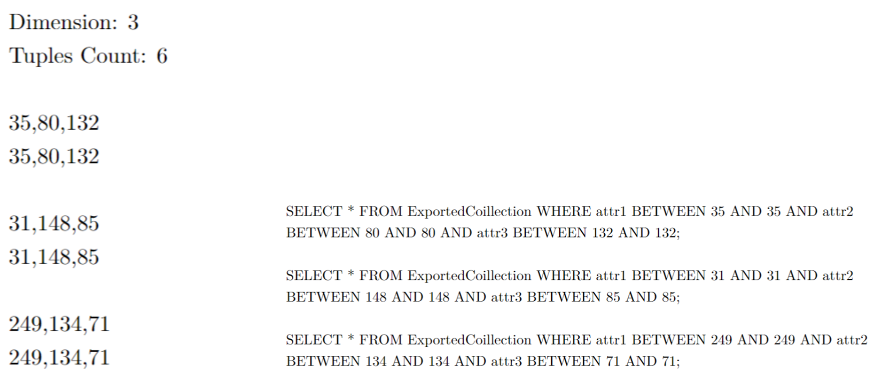
The file format depends on what we're exporting, if we choose to export tuples then the file format will be **.ctf**. On the other hand, if we choose to export queries, the file format will be **.qtf**. In addition, we can specify the file name by using the optional parameter **pCollectionName**.

6.6 Generate SQL Selects

There is also a public statistical function called **GenerateSQLSelects(char pCollectionName*)**, which generates SQL selects for the given query collection. This function takes the following parameters:

- **char pCollectionName***: name of collection and file, the default value for this optional parameter is **ExportedCoillection**. If no name is given then, the file will be named **qGeneratedSelects.sql**.

In Figure 6.4, we'll test this function, by generating SQL selects. The result can be seen in Figure 6.4b), for a given point query collection that is presented in Figure 6.4a.



```
Dimension: 3
Tuples Count: 6

35,80,132
35,80,132

31,148,85
31,148,85

249,134,71
249,134,71
```

```
SELECT * FROM ExportedCoillection WHERE attr1 BETWEEN 35 AND 35 AND attr2
BETWEEN 80 AND 80 AND attr3 BETWEEN 132 AND 132;

SELECT * FROM ExportedCoillection WHERE attr1 BETWEEN 31 AND 31 AND attr2
BETWEEN 148 AND 148 AND attr3 BETWEEN 85 AND 85;

SELECT * FROM ExportedCoillection WHERE attr1 BETWEEN 249 AND 249 AND attr2
BETWEEN 134 AND 134 AND attr3 BETWEEN 71 AND 71;
```

(a) Queries (b) Generated SQL Selects

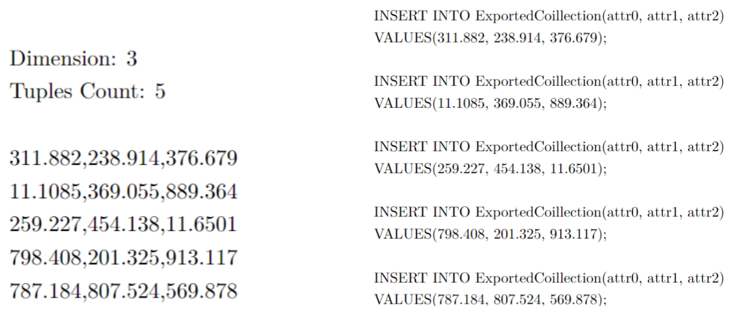
Figure 6.4: Example of Generated SQL Selects

6.7 Generate SQL Inserts

GenerateSQLInserts(char pCollectionName*) is a public statistical function that generates SQL inserts for a given data collection. This function takes the following parameters:

- **char pCollectionName***: name of collection and file, the default value for this optional parameter is **ExportedCoillection**. If no name is given then the file will be named **qGeneratedInserts.sql**.

In Figure 6.5, we will test this function. The generated SQL inserts can be seen in Figure 6.5a. The data collection from where they are generated is presented in Figure 6.5b.



```
Dimension: 3
Tuples Count: 5

311.882,238.914,376.679
11.1085,369.055,889.364
259.227,454.138,11.6501
798.408,201.325,913.117
787.184,807.524,569.878
```

(a) Generated Tuples

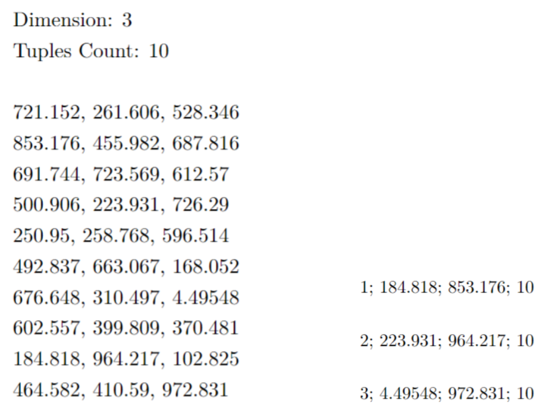
```
INSERT INTO ExportedCoillection(attr0, attr1, attr2)
VALUES(311.882, 238.914, 376.679);

INSERT INTO ExportedCoillection(attr0, attr1, attr2)
VALUES(11.1085, 369.055, 889.364);

INSERT INTO ExportedCoillection(attr0, attr1, attr2)
VALUES(259.227, 454.138, 11.6501);

INSERT INTO ExportedCoillection(attr0, attr1, attr2)
VALUES(798.408, 201.325, 913.117);

INSERT INTO ExportedCoillection(attr0, attr1, attr2)
VALUES(787.184, 807.524, 569.878);
```

(b) Generated SQL Inserts

Figure 6.5: Example of Generated SQL Inserts

6.8 Print Histogram

Print Histogram is another public statistical function that allows us to find out minimal, maximal and number unique values (values that appear only once in the collection) of each dimension in the collection. This function is called **PrintHistogram(uint pIntervalCount)** and it has one optional parameter which is **uint pIntervalCount** which allows us to specify a number of intervals, the default value of this parameters is zero which means that no intervals will be created. Only the minimal, maximal and number of unique values of each dimension in the collection will be printed. Also, we can save the histogram into a text file using the following function **PrintHistogram(char* pFileName, uint pIntervalCount)**. Which again is the same statistical function as **PrintHistogram(uint pIntervalCount)**, only that this function allows us to save the histogram into a file. The information for each dimension is printed in the following order: **dimension; minimal value; maximal value; number of unique values;**. In Figure 6.6, we'll test this function we can generate ten, three dimensional, random tuples and call the function PrintHistogram(char* pFileName).



```
Dimension: 3
Tuples Count: 10

721.152, 261.606, 528.346
853.176, 455.982, 687.816
691.744, 723.569, 612.57
500.906, 223.931, 726.29
250.95, 258.768, 596.514
492.837, 663.067, 168.052
676.648, 310.497, 4.49548
602.557, 399.809, 370.481
184.818, 964.217, 102.825
464.582, 410.59, 972.831
```

(a) Generated Tuples

```
1; 184.818; 853.176; 10

2; 223.931; 964.217; 10

3; 4.49548; 972.831; 10
```

(b) Result

Figure 6.6: Example of Generated Histogram

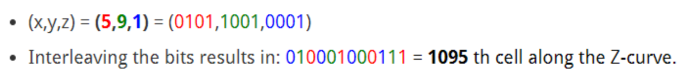
6.9 Z-Curve(Order)

In mathematical analysis and computer science, functions which are Z-order, Lebesgue curve, Morton space filling curve, Morton order or Morton code map multidimensional data to one dimension while preserving locality of the data points [18]. For this purpose a support function called **ZOrder(int pLeft, int pRight)** is implemented in **cCollectionGenerator**, which sorts the tuples according to the Z-Order. The function takes two parameters:

- **int pLeft**: from where to start the sorting, from which tuple index
- **int pRight**: to which tuple index the sorting will be done

This function allows us to represent multidimensional tuples to one dimension by generating appropriate *Z address* for each tuple. The addresses are generated by calling the function **GenerateZAddresses()**. **GenerateZAddresses()** generates *Z address* for each tuple in the collection by interleaving the bits of each element in the tuple.

For example If we want to convert a certain set of integer coordinates to a Morton code (Figure 6.7), we have to convert the decimal values to binary and interleave the bits of each coordinate [19], [20]:



- (x,y,z) = (**5**,**9**,**1**) = (0101,1001,0001)
- Interleaving the bits results in: 010001000111 = **1095** th cell along the Z-curve.

Figure 6.7: Example of Conversion of Tuple to Z-Address

Once we have *Z addresses* for every tuple in the collection, we can quick sort the addresses using quick sort algorithm and obtain a result like in the figure bellow (Figure 6.8). Note that the X and Y axis are flipped on the graph of this example.

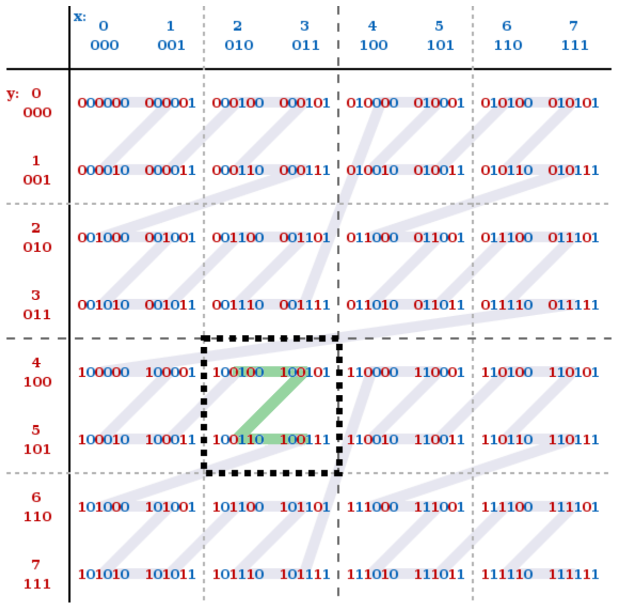


Figure 6.8: Example of Z-Order

When the order of the tuple is required. Firstly, the function **GenerateZAddresses()** has to be called which will generate *Z addresses* for all the tuples in the collection. Once we have the *Z addresses* for all the tuples we can call the function **ZOrder(int pLeft, int pRight)** and order the tuples according to their *Z addresses*. This function is set as private and used upon generating *range queries* (Section 5.5) and generating *cartesian range queries* (Section 5.6).

6.10 Other Useful Support Functions

1. **GetTuple(uint order)**: Allows us to obtain the tuple of the specified order. By specifying the order of the tuple into the parameter *pOrder*. For example if we specify zero, it will return the first tuple in the collection

2. **GetNextTupleFromFile()**: Allows us to obtain the next tuple from the file. This function is used when importing a data collection from a file

3. **GetNextTuple()**: Gives us an option to obtain the next tuple in the data collection that is stored inside the memory

4. **ResetPosition()**: Gives us an option resets the order of the tuples. For example if the last returned tuple was the fifth tuple in the collection. Then if we call this function, the order will reset and if we call **GetNextTuple()** the first tuple will be returned

5. **GetTupleCount()**: Allows us to obtain the number of tuples inside the tuple collection

6. **GetQueriesCount()**: Allows us to obtain the number of tuples inside the query collection

7. **GetDimension()**: Gives us an option to obtain the size of the dimension. For example if it's a 2D data collection, it will return 2

8. **SetDataSource(char* pFileName)**: Gives us an option to display the collection in the console, the collection is not saved in memory

9. **SetDataSource(uint pDimension, uint pTupleCount)**: Allows us to generate tuples according the specified size of dimension, number of tuples. Which is specified into the parameters *pDimension* and *pTupleCount*. The collection is displayed in the console, and it is not saved in memory

10. **SetDataSource(uint pDimension, uint pTupleCount, Distribution pDistribution)**: Allows us to generate tuples according the specified size of dimension, number of tuples, and distribution. By specifying them into the parameters *pDimension*, *pTupleCount*, and *pDimension*. The collection is displayed in the console, and it is not saved in memory

Chapter 7

Experiments

In this chapter we'll test the implementation of the new generator *cCollectionGenerator* by generating tuples, queries, performing sorts, shuffles, exporting data collections, generating SQL selects and inserts. Where the time will be measured in *seconds* (ms). All experiments are executed in a single thread on Intel(R) Core(TM) i7-8750H CPU with 16GB of RAM memory. Also, all experiments will be performed multiple times in order to get average time.

7.1 Experiments With Data Collections

In this experiment we'll generate one million tuples for the following dimensions: 2, 3, 5, 7, 10, 16, 24, 32, 64, 128. The experiment will be ran on the following distributions: Uniform, Normal and Logonormal. The result of the experiments can be seen in the Figure 7.1, where the measured times are plotted.

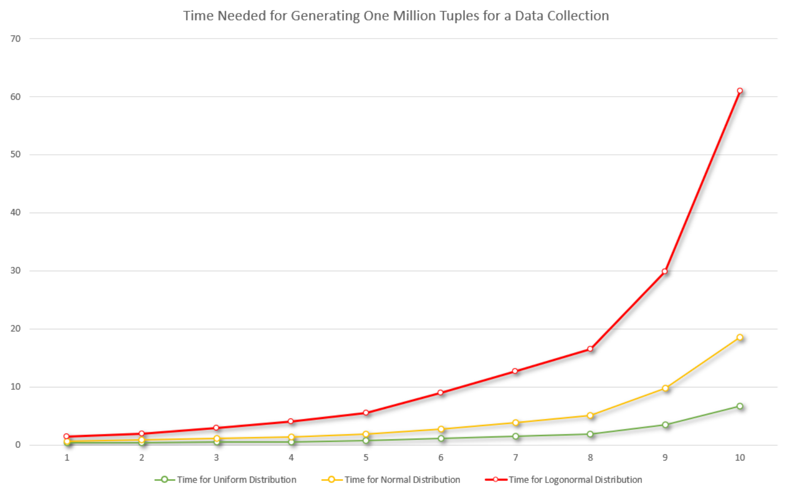


Figure 7.1: Experiments Performed on Data Collection

7.2 Experiments With Query Collections

In this experiment we'll generate ten thousand queries of the following dimension: 2, 16, 64, 128. The experiment will be ran on the following query generators: point query, partial match query and narrow range query generator. The result of the experiments can be seen in the Figure 7.2, where the measured times are plotted.

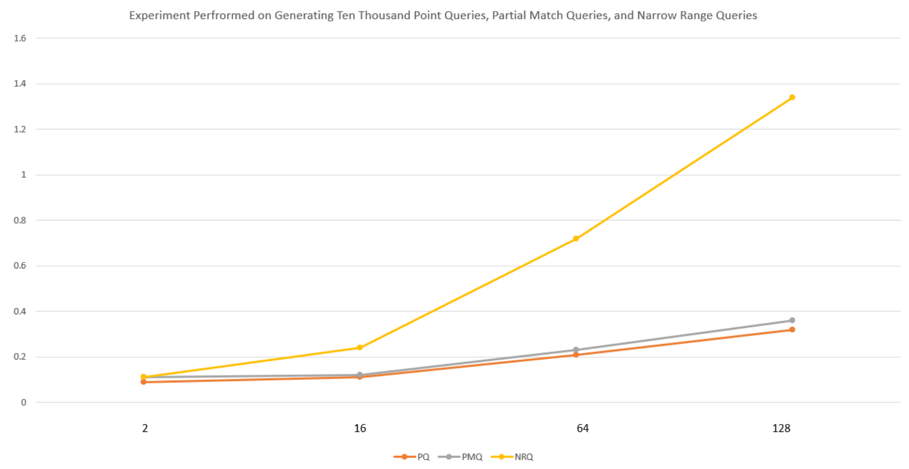


Figure 7.2: Experiments Performed on Query Collection

Also, we'll perform an experiment where we'll generate one thousand range queries. Where each query will return between two to five hundred tuples. The experiment will be ran again on the same dimensions.The result of the experiments can be seen in the Table 7.1.

Dimension	***Time for Range Queries *Measured on One Thousand Queries***
2	*12*
16	80
64	134.5
128	221.8

Table 7.1: Result of Performed Experiment on Generating Range Queries

7.3 Experiments With Statistical and Support Functions

In this experiment we'll sort a data collection by the Z-Order sort. Then we'll Generate a histogram for the data collection. Finally, we'll shuffle the data collection. The experiment will be performed with data collection of ten thousand tuples, from the following dimension: 2, 16, 64, 128. The results of the measurements can be seen in the Table 7.2.

Dimension	***Z-Order Sort***	***Generate Histogram***	***Shuffle***
2	0.15	0.07	0.000003
16	0.73	0.63	0.000004
64	0.26	2.61	0.000008
128	0.52	5.04	0.000011

Table 7.2: Experiments Performed With Statistical and Support Functions

Chapter 8

Conclusion

In this chapter we'll have a short discussion where we'll evaluate the implementation and its performance. Also, we'll sum up the work done, discuss what can be improved and what else can be implemented in the future.

A new and improved generator with various of futures available has been implemented. A comparison between the current generator *cTupleGenerator* and the new one *cCollectionGenerator* can be seen in the Table 8.1.

Available Futures	***cTupleGenerator***	***cCollectionGenerator***
Load Data from a Collection File	✓	✓
Generate Random Data	✓	✓
Available Distributions	**1**	**6**
Generate Various Types of Queries	×	✓
Sort Data Collection	×	✓
Shuffle Data Collection	×	✓
Export Data Collection	×	✓
Generate SQL Selects	×	✓
Generate SQL Insers	×	✓
Generate Histogram	×	✓

Table 8.1: Comparison Between *cTupleGenerator* and *cCollectionGenerator*

Based on the results of the experiments performed on a data collection, where data was generated by different types of distributions can be seen in the Table 8.2. The results are quite promising. However, we can see how the time increases when more complex distribution is used.

Dimension	***Uniform Distribution***	***Normal Distribution***	***Logonormal Distribution***
2	*0.43*	0.75	1.51
3	0.48	0.91	2
5	0.53	1.2	3
7	0.58	1.47	4.1
10	0.85	1.94	5.61
16	1.14	2.76	9
24	1.59	3.89	12.7
32	1.93	5.14	16.6
64	3.59	9.81	29.9
128	6.74	18.6	61

Table 8.2: Results of Experiments Performed on Data Collection

Then we can see the results of the experiments performed on different type of query collections, in the Table 8.3. Where *PQ* represents Point Query, *PMQ* represents Partial Matach Query, *NRQ* represents Narrow Range Query, and *RQ* represents Range Query. Here again the time is quite promising, however we can see how it increases when more complex type of queries are generated.

Dimension	***PQ***	***PMQ***	***NRQ***	***RQ***
2	*0.09*	0.1	0.11	12
16	0.11	0.12	0.24	80
64	0.2	0.23	0.72	134.5
128	0.32	0.36	1.34	221.8

Table 8.3: Results of Experiments Performed on Data Collection

Finally, to summarize the whole work. A new and improved generator has been implemented. However, this doesn't mean that there's no room for improvement. For example as future work, various types of sorts can be implemented. It will be quite useful to have more options for sorting the data collection. Also, some other distributions can be implemented and new approaches for generating queries can be proposed for studying and implementing. Furthermore, the generator can be optimized so it can work faster. There is a good potential in the new generator on which can be done various of new implementations.

Bibliography

1. CHOVANEC, Peter. *QuickDB – Yet Another Database Management.* 2015.
2. HOROWITZ, Ellis; SAHNI, Sartaj. *Fundamentals of Data Structures.* Ed. by HICKS, Robert Drew. Sung Kung Computer Book Co., 1987.
3. *B-Tree In Data Structure: Everything You Need to Know in Detail* [online]. 2021-09-16 [visited on 2022-04-27]. Available from: `https://www.simplilearn.com/tutorials/data-structure-tutorial/b-tree-in-data-structure`.
4. KUMAR, Ajitesh. *Dummies notes - what is B-tree and why use them?* 2015-05. Available also from: `https://vitalflux.com/dummies-notes-what-is-b-tree-and-why-use-them/`.
5. *Basics of R Tree* [online]. 2021-08-17 [visited on 2022-04-27]. Available from: `https://iq.opengenus.org/r-tree/`.
6. KATIYAR, Puloma; VU, Tin; MIGLIORINI, Sara; BELUSSI, Alberto; ELDAWY, Ahmed. *A Spatial Data Generator on the Web* [online]. 2020 [visited on 2022-04-16]. Available from: `https://spider.cs.ucr.edu/`.
7. *Simple Plot* [online] [visited on 2022-04-11]. Available from: `http://www.shodor.org/interactivate/activities/SimplePlot/`.
8. *Demos* [online] [visited on 2022-04-25]. Available from: `https://www.desmos.com/calculator`.
9. *Uniform real distribution* [online]. 2021-09-08 [visited on 2022-04-10]. Available from: `https://www.cplusplus.com/reference/random/uniform_real_distribution/`.
10. AHSANULLAH, Mohammad; KIBRIA, B. M. Golam; SHAKIL, Mohammad. Normal Distribution. In: *Normal and Student´s t Distributions and Their Applications.* Paris: Atlantis Press, 2014, pp. 7–50. ISBN 978-94-6239-061-4. Available from DOI: `10.2991/978-94-6239-061-4_2`.
11. *Normal Distribution* [online] [visited on 2022-04-17]. Available from: `https://www.cplusplus.com/reference/random/normal_distribution/`.
12. *Std::Lognormal$_d$istribution.* Available also from: `https://cplusplus.com/reference/random/lognormal_distribution/`.

13. KATIYAR, Puloma; VU, Tin; ELDAWY, Ahmed; MIGLIORINI, Sara; BELUSSI, Alberto. SpiderWeb: A Spatial Data Generator on the Web. In: *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*. Seattle, WA, USA: Association for Computing Machinery, 2020, pp. 465–468. SIGSPATIAL '20. ISBN 9781450380195. Available from DOI: `10.1145/3397536.3422351`.

14. *Sierpinski Triangle* [online] [visited on 2022-04-19]. Available from: `https://nrich.maths.org/4757`.

15. *Query* [online]. 2020-08-24 [visited on 2022-04-20]. Available from: `https://www.techopedia.com/definition/5736/query`.

16. KOSTAL, Lubomir; LANSKY, Petr; POKORA, Ondrej. Measures of statistical dispersion based on Shannon and Fisher information concepts. *Information Sciences*. 2013, vol. 235, pp. 214–223. ISSN 0020-0255. Available from DOI: `https://doi.org/10.1016/j.ins.2013.02.023`. Data-based Control, Decision, Scheduling and Fault Diagnostics.

17. PAGEL, Bernd-Uwe; SIX, Hans-Werner; TOBEN, Heinrich; WIDMAYER, Peter. Towards an Analysis of Range Query Performance in Spatial Data Structures. In: *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. Washington, D.C., USA: Association for Computing Machinery, 1993, pp. 214–221. PODS '93. ISBN 0897915933. Available from DOI: `10.1145/153850.153878`.

18. KILIMCI, Perihan; KALIPSIZ, Oya. Indexing of spatiotemporal Data: A comparison between sweep and z-order space filling curves. In: *International Conference on Information Society (i-Society 2011)*. 2011, pp. 450–456. Available from DOI: `10.1109/i-Society18435.2011.5978495`.

19. BAERT, Jeroen. *Morton encoding/decoding through bit interleaving* [online]. 2013-09-07 [visited on 2022-04-15]. Available from: `https://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/`.

20. SLAYTON, Zack. *Z-Order Indexing for Multifaceted Queries in Amazon DynamoDB: Part 1* [online]. 2017-05-17 [visited on 2022-04-17]. Available from: `https://aws.amazon.com/blogs/database/z-order-indexing-for-multifaceted-queries-in-amazon-dynamodb-part-1/`.