SAPIENZA
UNIVERSITÀ DI ROMA

Software Engineering

AY 2023/2024

Project: Anomaly Detection

Authors:

Dalla Ragione Ivan 2009768
Lombardi Lorenzo 2003512
Fragale Nicolò 2017378

Supervisor:

Tronci Enrico

Contents

1 Introduction **1**

1.1 General Description . . . 1

1.2 System and its environment: a visual representation . . . 1

2 User Requirements **2**

2.1 **Use-Case diagram** . . . 2

3 System Requirements **3**

3.1 **Diagrams** . . . 4

3.1.1 **System Architecture Diagram** . . . 4

3.1.2 **Activity Diagram** . . . 5

3.1.3 **State Diagram** . . . 6

3.1.4 **Message Sequence Chart** . . . 8

4 Implementation **9**

4.1 System Components . . . 9

4.1.1 **initRedisDataStream** . . . 9

4.1.2 **fillLogFromRedisStream** . . . 11

4.1.3 **dbModelCalc** . . . 13

4.1.4 **PostgreSql** . . . 16

4.1.5 **Redis** . . . 19

4.1.6 **Monitor** . . . 19

4.1.7 **Utilities** . . . 20

5 Results **22**

5.1 Datasets . . . 22

5.2 Performance . . . 25

5.2.1 Glass Identification performance . . . 26

5.2.2 wineQuality . . . 27

5.2.3 airQuality . . . 28

5.2.4 Comparison between datasets . . . 29

5.2.5 Performance Analysis with infinite tolerance . . . 30

1 Introduction

1.1 General Description

The project involves the development of an Anomaly Detection system for data streams.
Given a configurable time window **w**, the system detects all the values that differ significantly from the average of the model .

1.2 System and its environment: a visual representation

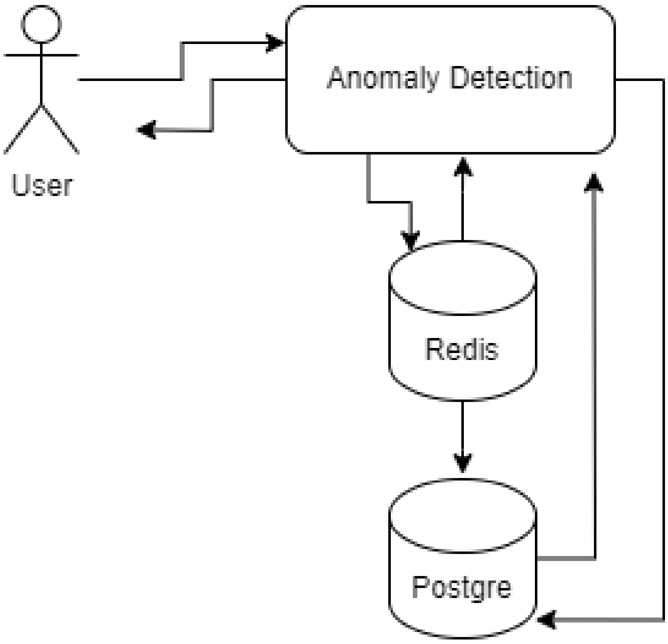


Figure 1: System environment

2 User Requirements

The user requirements and the respective use-case diagrams are listed below

1. Insert the data to analyze in a CSV file
2. Choose a window
3. Choose the tolerance in percentage
4. Choose the fields to analyze
5. Calculate the average
6. Calculate the covariance
7. Observe the anomalies detected

2.1 Use-Case diagram

The use case diagram for the previously listed requirements is illustrated below:

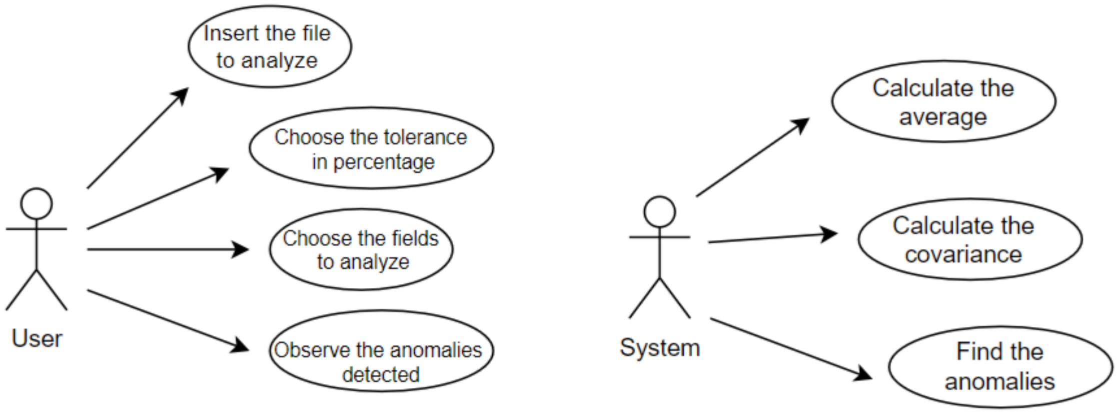

Insert the file to analyze
Choose the tolerance in percentage
Choose the fields to analyze
Observe the anomalies detected
User
Calculate the average
Calculate the covariance
Find the anomalies
System

Figure 2: Use-case diagram

3 System Requirements

The system requirements and the respective diagrams are listed below

1. initRedisDataStream
2. fillLogFromRedisStream
3. dbModelCalc
4. Utilities:
   (a) ReplacerCSV
   (b) randomLog
5. PostgreSQL
6. Redis

The previous system requirements will be explained in detail in the following parts

3.1 Diagrams

3.1.1 System Architecture Diagram

The system architecture diagram wants to show the interactions between the systems process.

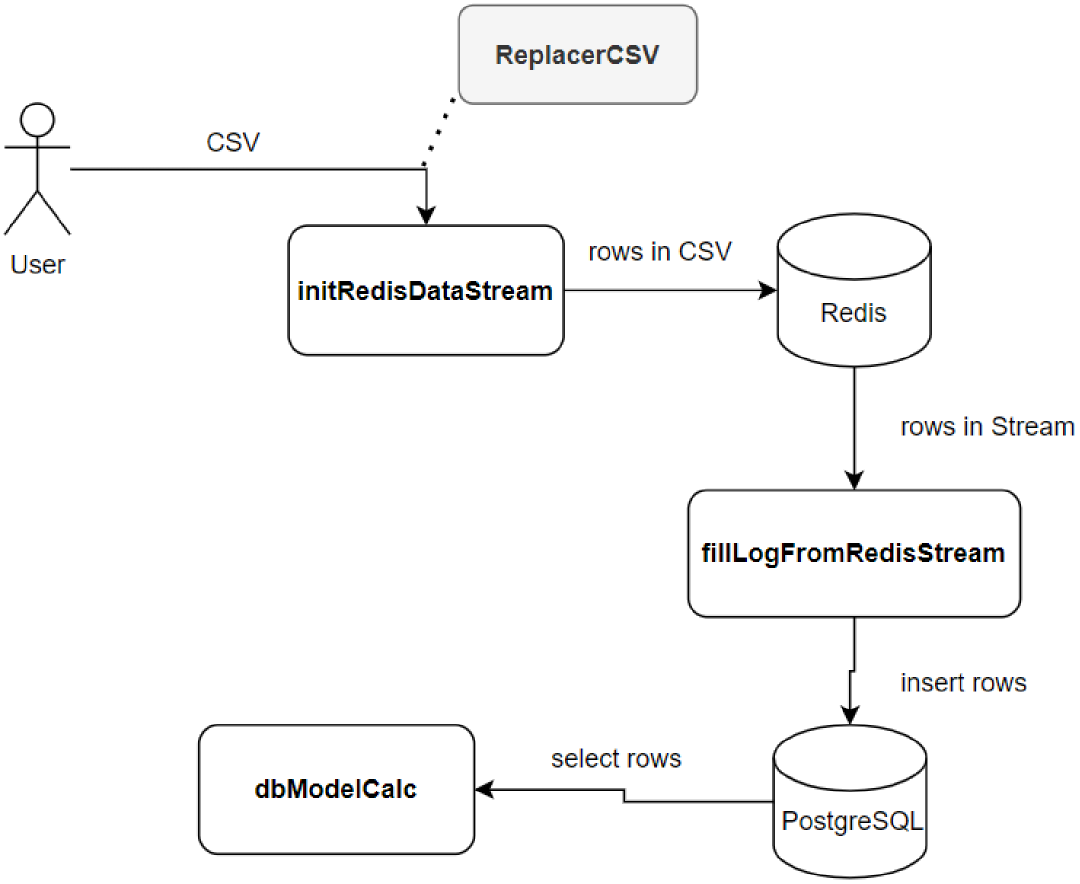


Figure 3: System Architecture Diagram

3.1.2 Activity Diagram

The following activity diagram is focused on anomaly finder flow, showing the execution decision making.

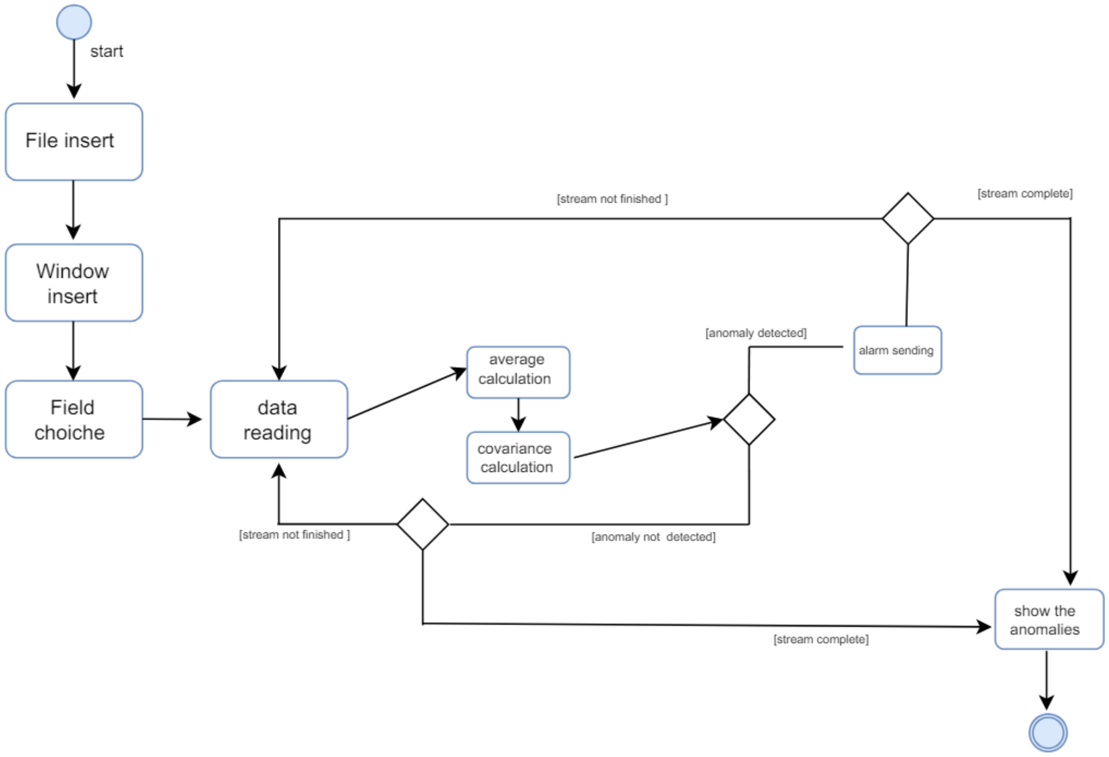


Figure 4: Activity Diagram

3.1.3 State Diagram

System State Diagram

The system state diagram wants to show the sequential execution flow, showing the action performed for every state and the event that determines the state transition.

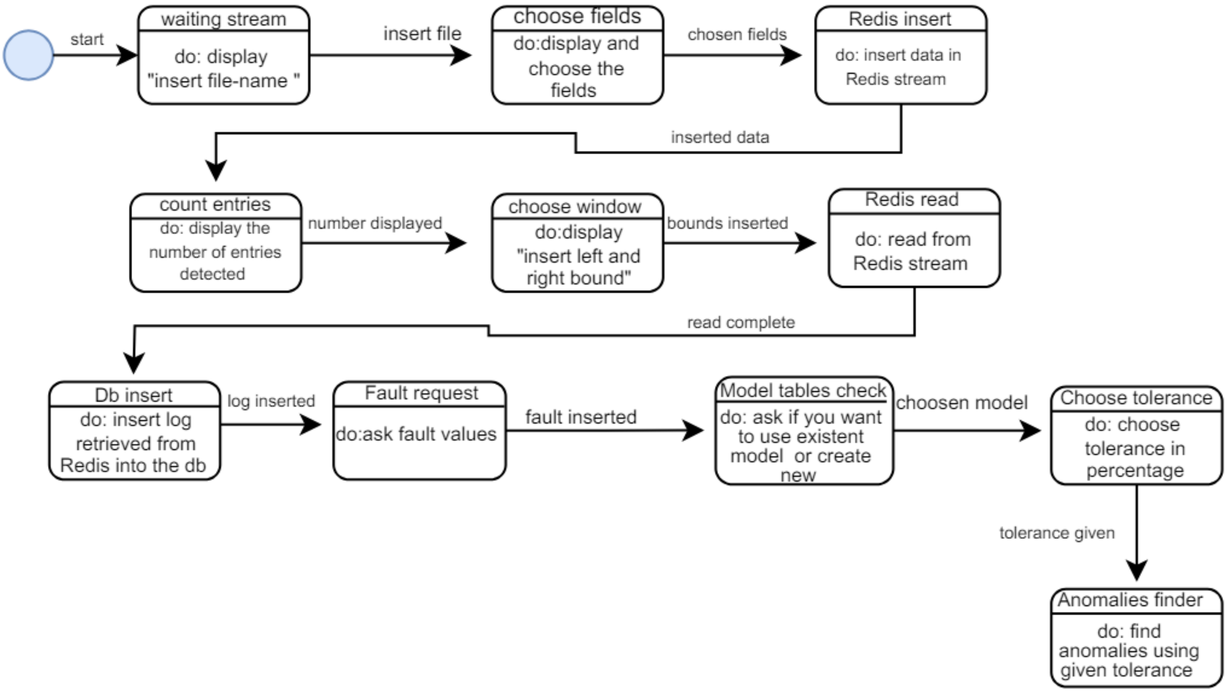

start
waiting stream
do: display "insert file-name "
insert file
choose fields
do:display and choose the fields
chosen fields
Redis insert
do: insert data in Redis stream
inserted data
count entries
do: display the number of entries detected
number displayed
choose window
do:display "insert left and right bound"
bounds inserted
Redis read
do: read from Redis stream
read complete
Db insert
do: insert log retrieved from Redis into the db
log inserted
Fault request
do:ask fault values
fault inserted
Model tables check
do: ask if you want to use existent model or create new
choosen model
Choose tolerance
do: choose tolerance in percentage
tolerance given
Anomalies finder
do: find anomalies using given tolerance

Figure 5: State Diagram

AnomalyFinder State Diagram

This diagram is a view of the AnomalyFinder state. It shows the particular details of the anomalies research

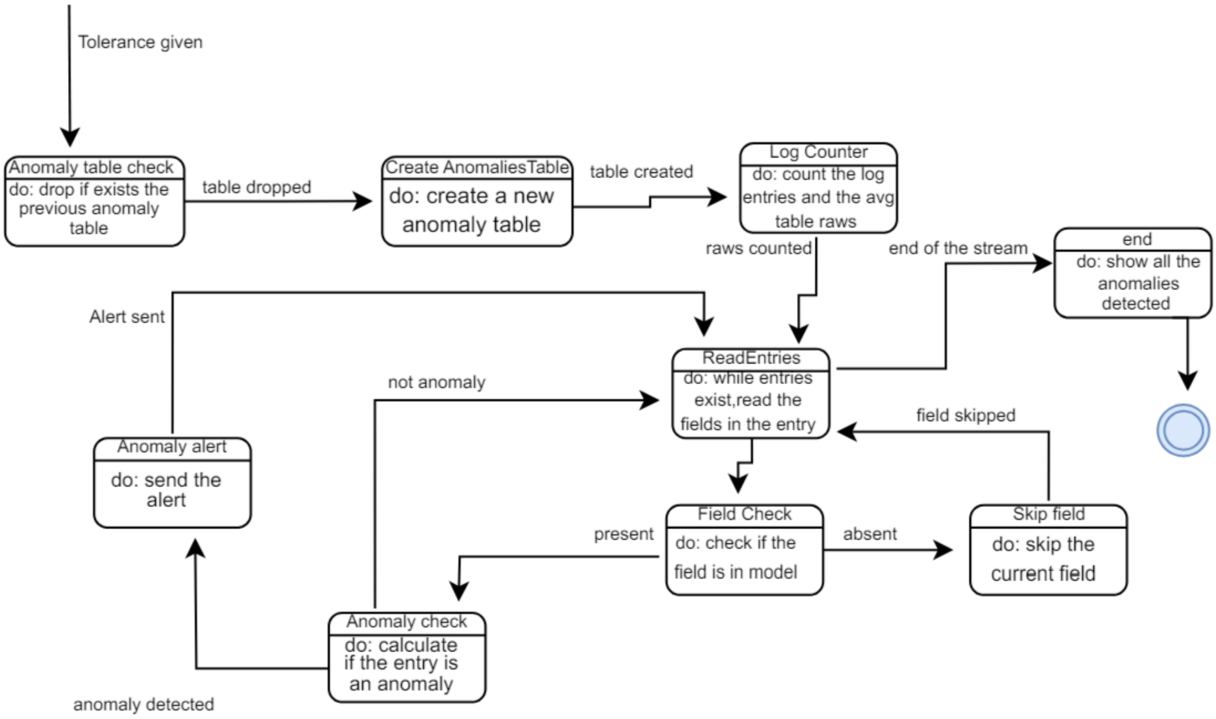

Tolerance given
Anomaly table check
do: drop if exists the previous anomaly table
table dropped
Create AnomaliesTable
do: create a new anomaly table
table created
Log Counter
do: count the log entries and the avg table raws
raws counted
end of the stream
end
do: show all the anomalies detected
Alert sent
ReadEntries
do: while entries exist,read the fields in the entry
not anomaly
field skipped
Anomaly alert
do: send the alert
Field Check
do: check if the field is in model
present
absent
Skip field
do: skip the current field
Anomaly check
do: calculate if the entry is an anomaly
anomaly detected

Figure 6: State Diagram AnomalyFinder

3.1.4 Message Sequence Chart

This message sequence chart diagram wants to highlight the interaction and the messages between the first two components of our program: initRedisDataStream and fillLogFromRedisStream, (they're written shortly in the diagram to make it more readable). It's possible to see also the user, which is an active actor for the execution of our two components, Redis and PostgreSQL, used to stream and store our data

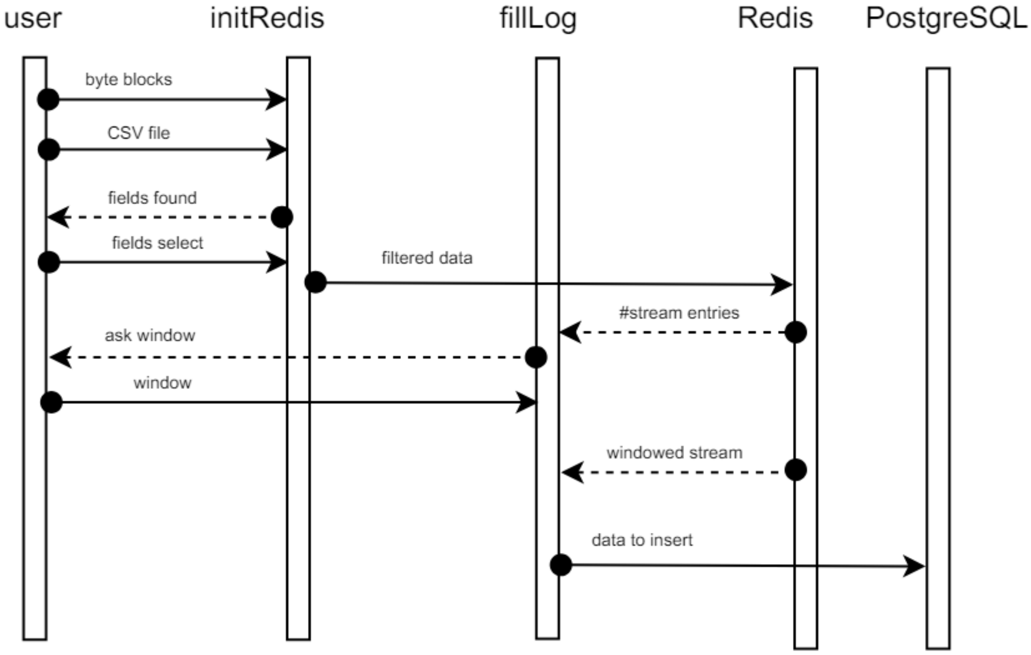


Figure 7: Message Sequence Chart diagram

4 Implementation

4.1 System Components

For each component, the functions that compose it will be briefly explained and for the important ones, the pseudocode will be shown

4.1.1 initRedisDataStream

This component reads from a CVS file, allows to choose the byte size of fields and uploads the CSV rows in a Redis Stream.

- **buildLine**: this function accepts a character buffer (‘char*‘) called *buffer* and a pointer to a string vector (‘std::vector<std::string>&‘) called *fields*.If the vector contains an empty string, it’ll be removed otherwise it splits the string.

- **printLine**: given a vector of string *fields* in input, the strings are concatenated in *row* and printed (useful in debugging purpose).

- **readLine** : this function accepts 3 parameters: a pointer to a character array *buffer* where the read rows is stored, the buffer dimension in *buffer_size* and a pointer to FILE called *file* that represents the file where the row is read the row. ‘fgets‘ reads a rows from *file* and store it in *buffer*, only after the new line character removal. The maximum number of characters that can be red is specified in *buffer_size*.

- **stringToChar**: accepts a string ‘std::string‘ and converts it into a character array (‘char*‘).

- **exclusionCalc**: given a string vector *current_row* it returns an integer vectors ‘vector<int>‘ called *vect* that contains the index of the fields to analyze

- **excludeElements** accepts two parameters : *current _row*, wich is a string vector and represents the current row and *exclude_indices*, and integer vector that contains the fields to be not analyzed. The function filters the elements of the current row to return a new string vector called *result* that contains the right elements that the user wants to analyze.

- **createTable**: the function accepts 3 arguments: a *Con2DB* object that represents the connection with the database, *name* that represents the table name that we want to create and a string vector *fields* that contains the name of the fields in the table. The main goal of the functions is to create a table in the db with the specified name and the specified fields, if the table already exists, it is eliminated before the creation of the new one.

- **checkTable**: the function executes an SQL query to verify if a table with the specified name exists in the database and returns true if it exists.

- **dropTable**: the function executes an SQL query to delete the table with the specified name into the database

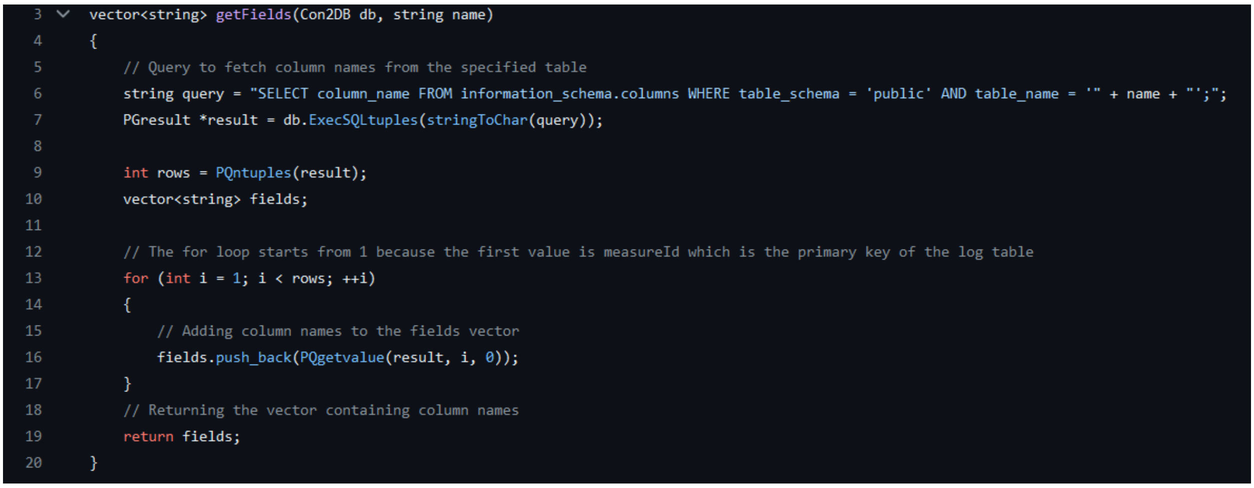
- **main**: initialize the program calling the function *init* with the arguments from the command line, it ends returning the *init* value

- **init**: the function initializes the entire program, establishes a connection with a Redis server checking that the stream is empty, then reads a particular CSV inserted by the user and process one row at time removing some unwanted fields if the user explicits them, and sends the right data to the stream. Initiaize the log table in the database. In the end it closes all the connections and returns 0

4.1.2 fillLogFromRedisStream

this component initializes the database table, requests the window to the user and reads the Redis stream inserting the rows in the database

- **split**: accepting a string in input, the function divides it into a substring separated by space and replaces the commas with a point in every substring. At the end, a string vector that contains all the substrings is returned
- **windowSelect**: the function asks to the user to choose a start and an endpoint, verifying if the values can be correctable used. If there is an error, the system tells the user to insert a valid input.
- **getFields**: the function extracts, using a sql query, the fields name (columns name) contained in the table *name*



```
vector<string> getFields(Con2DB db, string name)
{
    // Query to fetch column names from the specified table
    string query = "SELECT column_name FROM information_schema.columns WHERE table_schema = 'public' AND table_name = '" + name + "';";
    PGresult *result = db.ExecSQLtuples(stringToChar(query));

    int rows = PQntuples(result);
    vector<string> fields;

    // The for loop starts from 1 because the first value is measureId which is the primary key of the log table
    for (int i = 1; i < rows; ++i)
    {
        // Adding column names to the fields vector
        fields.push_back(PQgetvalue(result, i, 0));
    }
    // Returning the vector containing column names
    return fields;
}
```

Figure 8: getFields function

- **insertDb**: accepts four argouments: a *Con2DB* object, a string vector *fields* that contains the field's name of the table, a string *tableName* that specifies the table name where the data need to be inserted and a string *data* that specifies the values to insert. The function wants to build a query using the field's name given and the data. Error handler is present to manage errors

- **reader**: this function manages the Redis message flow and its insertion into the database. Allowing the user to choose a window if requested. We can divide the function into some steps: 1) Start by establishing a connection to Redis and getting the field names of the "log" table from the database. 2) Sets the read lock for the Redis read feature and, if enabled, allows the user to select a data window. 3) Enters an infinite loop to read messages from the Redis stream using the 'XREADGROUP' function. 4) For each message read, it checks whether it should be inserted into the database table according to the selected window and inserts data if necessary. 5) Increment the read message counter and free the memory allocated for the message. 6)At the end close the connection to Redis and end the function
- **redisUtils**: defines two functions called *getKey* and *getValue* where both include a pointer to *redisReply* and an integer as arguments. Both access the Redis response data structure pointed to by 'reply' and get the first element. Of the latter, they access the first element by considering the element in position *number*. *getKey* accesses this element's string array and returns the first string (element 0), which represents the key. *getValue* accesses this element's string array and extracts the second element, which represents the value associated with the key. Next, it accesses the string array of this

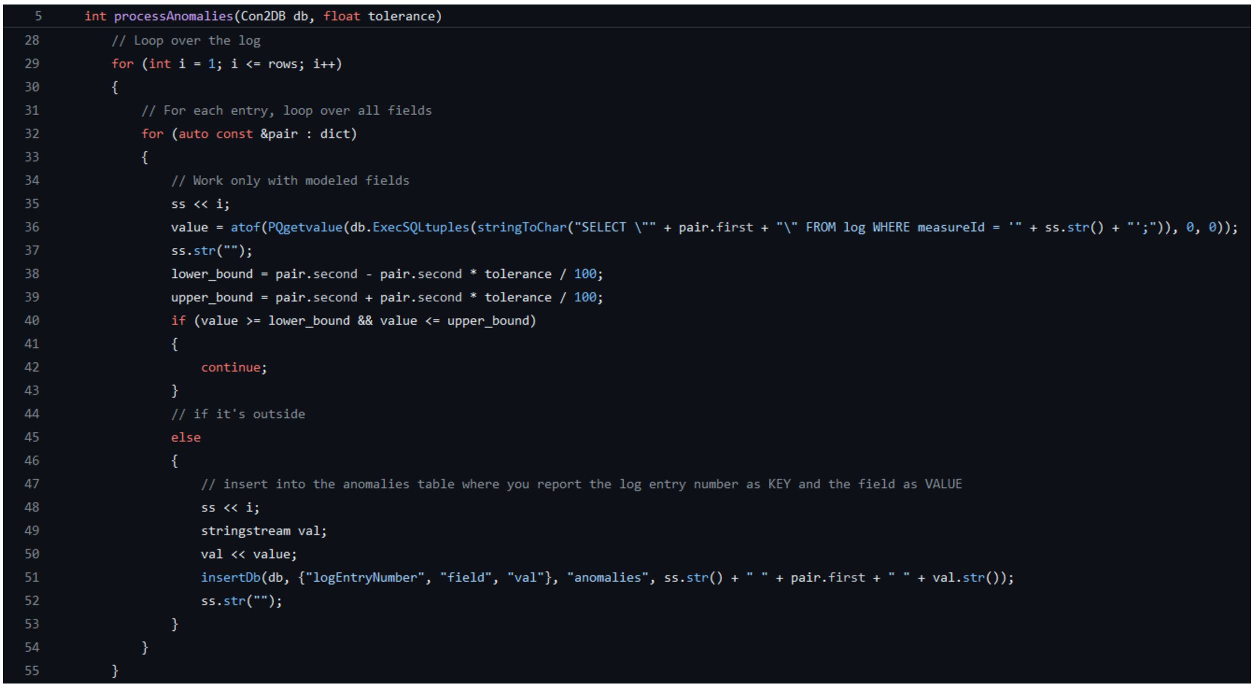
value and returns the second element, which contains the actual stored value.

4.1.3 dbModelCalc

This component uses the log table and a possible and existent model/new model to calculate average, covariance. It asks for the tolerance percentage and it finds the anomalies.

- **wasteCalc**: allows the user to insert special values used to indicate a waste or error
- **covDropAsk**: allows the user to choose the old covariance table (if it exists) or create a new one.
- **selectTolerance**: allows the user to select a particular tolerance in percentage. The default is set to 50%
- **modifiedFind**: the function looks into the vector for an item to find called *to_find*. It returns true if the element is found, false otherwise.
- **initiAvgModel**: initializes a model for the average calculus into the database. The function retrieves all the fields and their values from the log. Then the average is calculated and is returned the value 0, which indicates the end of the model creation
- **initCovModel**: it creates the covariance tables in the database. It starts doing a query and obtaining the number of rows in the log table and in the covariance table. For each field, the function creates a covariance table if it does not exist yet
- **processAnomalies**: for each row in log, the function verifies if the current value is included into the tolerance boundaries.

If is not, it’s put into the anomalies table.



```
int processAnomalies(Con2DB db, float tolerance)
    // Loop over the log
    for (int i = 1; i <= rows; i++)
    {
        // For each entry, loop over all fields
        for (auto const &pair : dict)
        {
            // Work only with modeled fields
            ss << i;
            value = atof(PQgetvalue(db.ExecSQLtuples(stringToChar("SELECT \"" + pair.first + "\" FROM log WHERE measureId = '" + ss.str() + "';")), 0, 0));
            ss.str("");
            lower_bound = pair.second - pair.second * tolerance / 100;
            upper_bound = pair.second + pair.second * tolerance / 100;
            if (value >= lower_bound && value <= upper_bound)
            {
                continue;
            }
            // if it's outside
            else
            {
                // insert into the anomalies table where you report the log entry number as KEY and the field as VALUE
                ss << i;
                stringstream val;
                val << value;
                insertDb(db, {"logEntryNumber", "field", "val"}, "anomalies", ss.str() + " " + pair.first + " " + val.str());
                ss.str("");
            }
        }
    }
```

Figure 9: process anomalies function

- **dropTable**: this function is used to delete a particular database table.

- **dropAllCov**: this function is used to delete all the covariance tables into the database.

- **getKeysCov**: it retrieves the keys from the specified covariance table accepting two arguments: an object ‘Con2DB‘,(the connection to the database object), and a string ‘name‘, that is the name of the covariance table.. An SQL query is built to select the keys from the specified table and then execute using ‘ExecSQLtuples‘. At last, the function adds every key to a string vector that will be used in the future to analyze the obtained data

- **main**: is the core of the component, it manages all the main functions, it deals with :

- Establish a database connection
- Verify if exists is a log table yet
- Count the entries in the log table
- Waste values management
- Average and covariance tables creation
- Anomalies detection and software closure

4.1.4 PostgreSql

This section describes the database, its er-scheme and in particular, the single tables of the database.

Log

The first table is called Log. Here are stored all the logs from the CSV file. Every Log has an identifier (**measureId**: integer), a date(**date**: varchar) and time (**time**: varchar), and a dynamic number of attributes that represent the fields in the CSV file. Given that each CSV can potentially contain a different number of fields, we decided to manipulate this problem by building a table tailored to every different CVS that the software reads.

Anomaly

The second table is called anomaly, here we store the logs that present an anomaly in a particular field.

Every anomaly has an identifier (**measureId**: integer) specific to this table, the number that indicates its code in the Log table(**logEntryNumber**: integer(fk) references to measureId(Log)), the field(**field**: varchar) where the anomaly was caught and the value (**value**: varchar) identified as anomaly

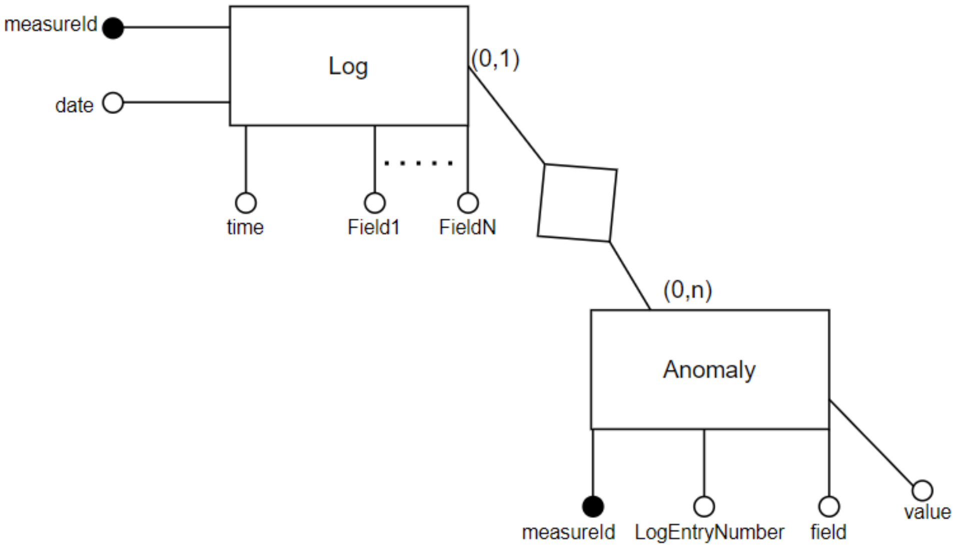


Figure 10: Log to Anomaly relation

Now we will continue with the other tables that are not in relationship with the previous two.

Average

The average table is used to create, for each value, the average which will be used to search for anomalies. Is composed of an identifier(**measureId**:int), a key(key:varchar) that indicates the field, and an average value (**average**:varchar). So, during the execution, the software accesses the average value for the specific key and checks if the current entry is an anomaly for a field.

Cov*Field*

The idea is to create a covariance table for each field composed by a key(**key**:varchar), that represents the field on which the covariance is calculated and a covariance value (covariance:varchar) that represents the specific value obtained calculating the covariance between the key and the specific field of the table Every table of this type, its call CovName, where name is a specific field.

Now we want to show the entire database, including the tables that are not connected with the others.

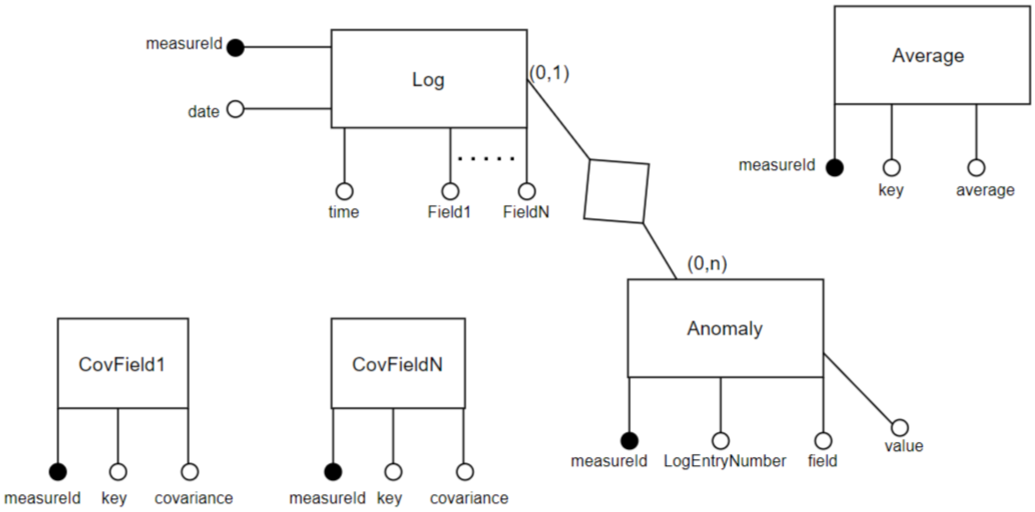


Figure 11: er-scheme

As previously explained, the fields can be variable and consequentially, the CovField tables can be variable.

4.1.5 Redis

We use Redis Stream and its c++ API to create a FIFO pipeline to allow message exchange between processes. The main Redis command used are:

- XTRIM: to clean the pipe
- XADD: to add messages into the pipe
- XLEN: to check the number of messages in the pipe
- XREADGROUP: to read the messages

In the first component: initRedisDataStream we use Redis to add our CSV rows in a pipe, it's possible to observe this action performed in the init function
In the second one : fillLogFromRedisStream we use Redis to insert the data into the database, it's possible to observe this action performed in the init function

4.1.6 Monitor

We implemented 5 different monitors, which can be used to retrieve different information:

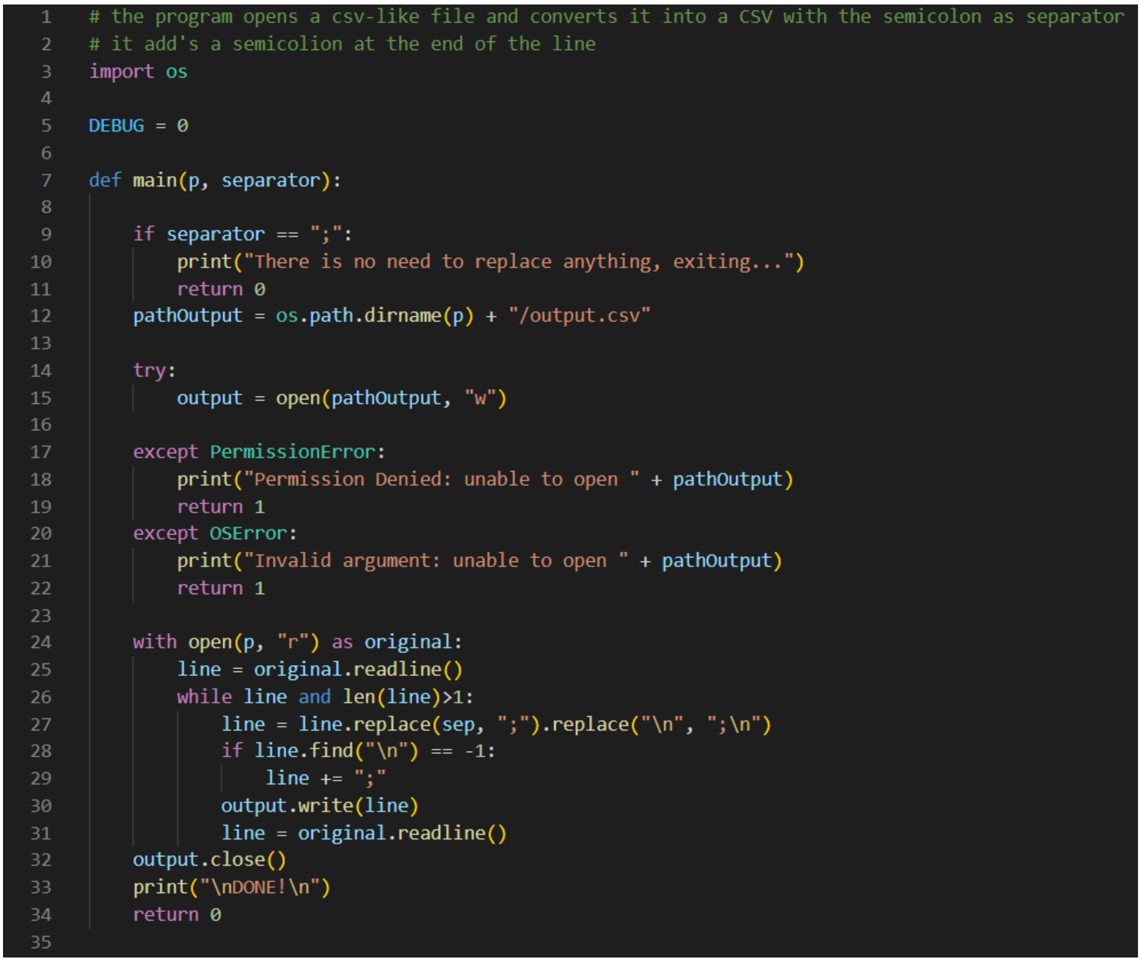
- XLENmonitor
- PRINTLOGmonitor
- PRINTAVGmonitor
- PRINTCOVmonitor
- PRINTANOMALIESmonitor

The full monitor documentation is written in the specific document

4.1.7 Utilities

These components are external scripts that are useful in software usage, they help the user to execute and test it.

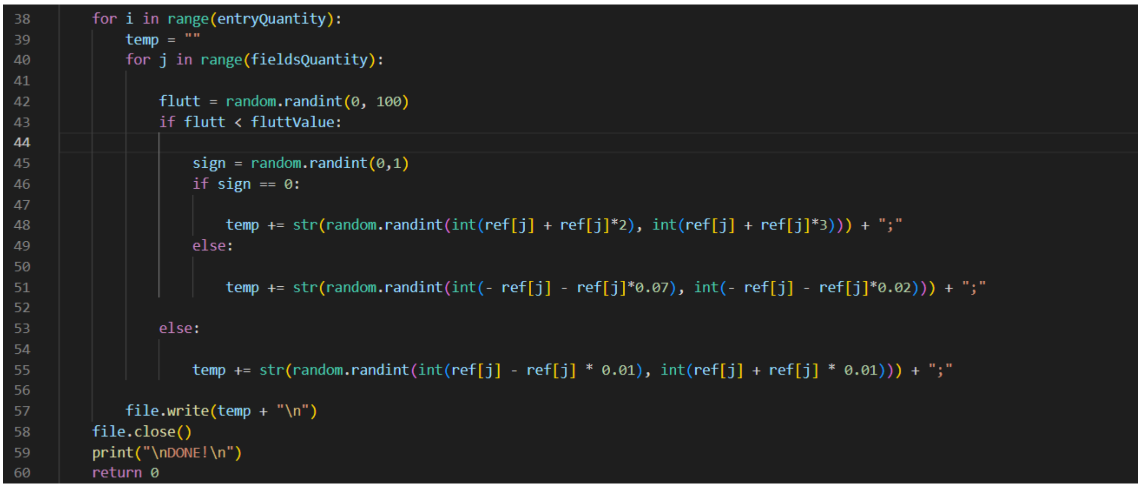
- **ReplacerCSV** : this file is used to modify a CSV file given in input. The software is created to be used with different CSV files, but there isn't a single standard that defines the separator between fields. Originally the fields were separated by a comma but actually, lots of online datasets use a semicolon. This replacer is developed to accept any CSV with any type of separator. The user can insert the separator and the script will convert into a semicolon



```python
# the program opens a csv-like file and converts it into a CSV with the semicolon as separator
# it add's a semicolion at the end of the line
import os

DEBUG = 0

def main(p, separator):

    if separator == ";":
        print("There is no need to replace anything, exiting...")
        return 0
    pathOutput = os.path.dirname(p) + "/output.csv"

    try:
        output = open(pathOutput, "w")

    except PermissionError:
        print("Permission Denied: unable to open " + pathOutput)
        return 1
    except OSError:
        print("Invalid argument: unable to open " + pathOutput)
        return 1

    with open(p, "r") as original:
        line = original.readline()
        while line and len(line)>1:
            line = line.replace(sep, ";").replace("\n", ";\n")
            if line.find("\n") == -1:
                line += ";"
            output.write(line)
            line = original.readline()
    output.close()
    print("\nDONE!\n")
    return 0

```

Figure 12: Replacer code

- **randomLog** : this file is used to create a fictitious CSV file to test the software. The user can personalize the file by choosing the number of entries and the number of fields per entry. The software works as follows:
  - ∗ after the user's choice, the software fills the first entry fields values with a random int
  - ∗ for each other entry, for each other field in the entry, the software generates different values using a flutt value that can be changed. The standard value is 5, which means that the 5% of times the range for the random value creation is bigger and the probability of obtaining an anomaly increases. This value can be set by modifying the global variable fluttValue
  - ∗ at the end of the previous steps, a text file is filled with all the entries and fields. Then the file is closed.



```
    for i in range(entryQuantity):
        temp = ""
        for j in range(fieldsQuantity):

            flutt = random.randint(0, 100)
            if flutt < fluttValue:

                sign = random.randint(0,1)
                if sign == 0:

                    temp += str(random.randint(int(ref[j] + ref[j]*2), int(ref[j] + ref[j]*3))) + ";"
                else:

                    temp += str(random.randint(int(- ref[j] - ref[j]*0.07), int(- ref[j] - ref[j]*0.02))) + ";"

            else:

                temp += str(random.randint(int(ref[j] - ref[j] * 0.01), int(ref[j] + ref[j] * 0.01))) + ";"

        file.write(temp + "\n")
    file.close()
    print("\nDONE!\n")
    return 0
```

Figure 13: RandomLog core portion of code

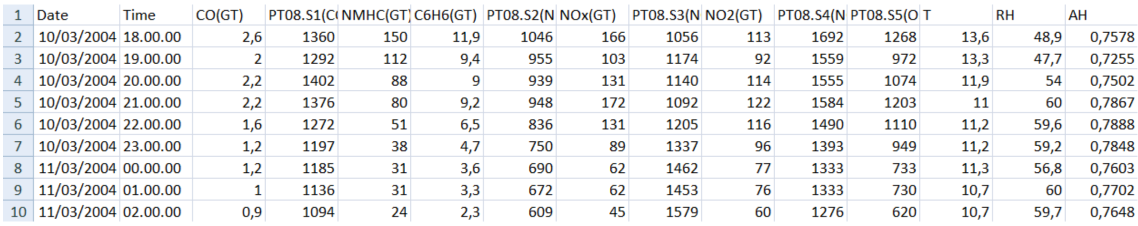
5 Results

In this section,the datasets and the performance results will be showed and explained.

5.1 Datasets

The Anomaly Detection system is tested using 3 different datasets selected from this repository, the datasets are:

- **airQuality**
  The dataset, contains 9358 istances of hourly averaged responses from an array of 5 metal oxide chemical sensors embedded in an Air Quality Chemical Multisensor Device. The device was located on the field in a significantly polluted area, at road level,within an Italian city.



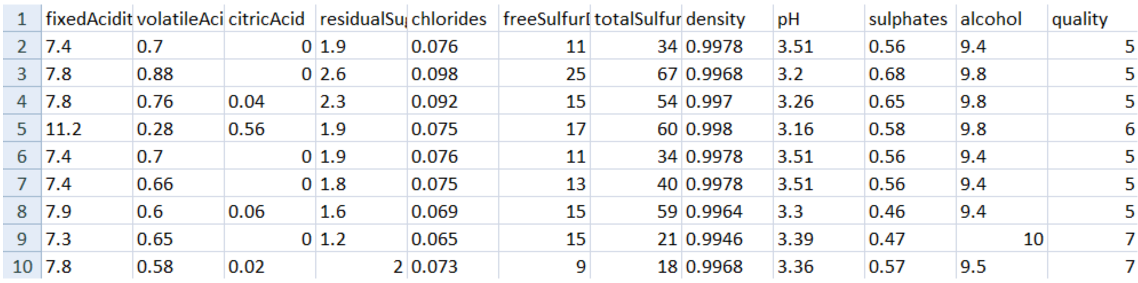
1	Date	Time	CO(GT)	PT08.S1(C	NMHC(GT	C6H6(GT)	PT08.S2(N	NOx(GT)	PT08.S3(N	NO2(GT)	PT08.S4(N	PT08.S5(O	T	RH	AH
2	10/03/2004	18.00.00	2,6	1360	150	11,9	1046	166	1056	113	1692	1268	13,6	48,9	0,7578
3	10/03/2004	19.00.00	2	1292	112	9,4	955	103	1174	92	1559	972	13,3	47,7	0,7255
4	10/03/2004	20.00.00	2,2	1402	88	9	939	131	1140	114	1555	1074	11,9	54	0,7502
5	10/03/2004	21.00.00	2,2	1376	80	9,2	948	172	1092	122	1584	1203	11	60	0,7867
6	10/03/2004	22.00.00	1,6	1272	51	6,5	836	131	1205	116	1490	1110	11,2	59,6	0,7888
7	10/03/2004	23.00.00	1,2	1197	38	4,7	750	89	1337	96	1393	949	11,2	59,2	0,7848
8	11/03/2004	00.00.00	1,2	1185	31	3,6	690	62	1462	77	1333	733	11,3	56,8	0,7603
9	11/03/2004	01.00.00	1	1136	31	3,3	672	62	1453	76	1333	730	10,7	60	0,7702
10	11/03/2004	02.00.00	0,9	1094	24	2,3	609	45	1579	60	1276	620	10,7	59,7	0,7648

Figure 14: airQuality entries

  These are the first ten rows of the CSV file, how it's possible to see,each row is composed of 15 fields:
  the date and the time of the sampling, and all the sensors that sample the Non Metanic HydroCarbons concentration, Benzene,CO...
  The dataset can be downloaded here, where also it's possible to find the fields documentation

- **wineQuality**

  The dataset contains two sub-dataset for an amount of 4898 istances. They are related to red and white variants of the Portuguese "Vinho Verde" wine.Due to privacy and logistic issues, only physicochemical (inputs) and sensory (the output) variables are available (e.g. there is no data about grape types, wine brand, wine selling price, etc.). In particular, the test was executed on the red wine sub dataset



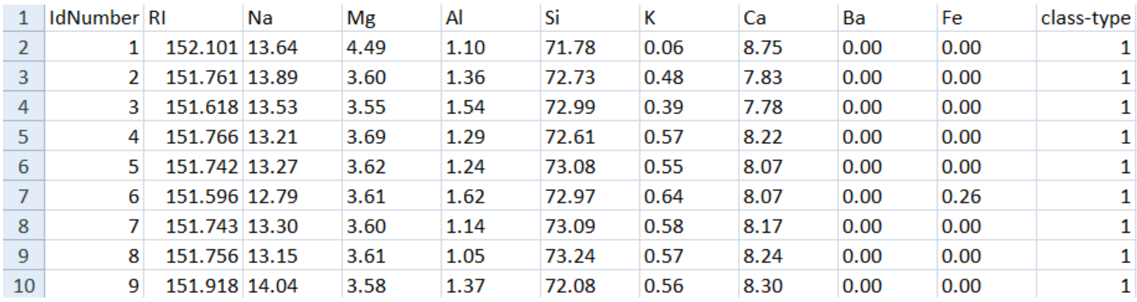
1	fixedAcidit	volatileAci	citricAcid	residualSu	chlorides	freeSulfur	totalSulfur	density	pH	sulphates	alcohol	quality
2	7.4	0.7	0	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5
3	7.8	0.88	0	2.6	0.098	25	67	0.9968	3.2	0.68	9.8	5
4	7.8	0.76	0.04	2.3	0.092	15	54	0.997	3.26	0.65	9.8	5
5	11.2	0.28	0.56	1.9	0.075	17	60	0.998	3.16	0.58	9.8	6
6	7.4	0.7	0	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5
7	7.4	0.66	0	1.8	0.075	13	40	0.9978	3.51	0.56	9.4	5
8	7.9	0.6	0.06	1.6	0.069	15	59	0.9964	3.3	0.46	9.4	5
9	7.3	0.65	0	1.2	0.065	15	21	0.9946	3.39	0.47	10	7
10	7.8	0.58	0.02	2	0.073	9	18	0.9968	3.36	0.57	9.5	7

Figure 15: wineQuality entries

  These are the first ten rows of the red wine CSV file, how it's possible to see,each row is composed of 11 fields that describe chemical properties of the wine.
  The dataset can be downloaded here, where also it's possible to find the fields documentation

- **Glass Identification**

  The dataset contains 214 entries, is the smallest of our tests, it's used in forensic science and describes the 6 types of glass, defined in terms of their oxide content. Each row is composed of 9 fields, that describe: refractive index, sodium, magnesium, aluminum, silicon, potassium etc...



1	IdNumber	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	class-type
2	1	152.101	13.64	4.49	1.10	71.78	0.06	8.75	0.00	0.00	1
3	2	151.761	13.89	3.60	1.36	72.73	0.48	7.83	0.00	0.00	1
4	3	151.618	13.53	3.55	1.54	72.99	0.39	7.78	0.00	0.00	1
5	4	151.766	13.21	3.69	1.29	72.61	0.57	8.22	0.00	0.00	1
6	5	151.742	13.27	3.62	1.24	73.08	0.55	8.07	0.00	0.00	1
7	6	151.596	12.79	3.61	1.62	72.97	0.64	8.07	0.00	0.26	1
8	7	151.743	13.30	3.60	1.14	73.09	0.58	8.17	0.00	0.00	1
9	8	151.756	13.15	3.61	1.05	73.24	0.57	8.24	0.00	0.00	1
10	9	151.918	14.04	3.58	1.37	72.08	0.56	8.30	0.00	0.00	1

Figure 16: glass identification entries

These are the first ten rows of the CSV file, how it's possible to see,each row is composed of 11 fields previously descripted. The dataset can be downloaded here, where also it's possible to find the fields documentation

5.2 Performance

For each dataset, we performed 3 types of measurements, using three different tolerance settings: 30%, 50%, 80%.
This is useful to observe how the time changes in a directly proportionate way to the growth of the entries, but in an inversely proportionate way to the growth of the tolerance percentage.
The time sampled comes from the execution time of the function *processAnomalies*
In particular, we tested our software using 3 different datasets:

- Glass Identification
- Wine Quality
- Air quality

5.2.1 Glass Identification performance

The first dataset tested is Glass Identification, the smallest one. We did 100 measurements for each tolerance percentage (all the measurement values for each tolerance are available in the output file in the directory) and we obtain this information:

tolerance	**anomalies**	**average time(s)**
30%	1026	2.978552009
50%	802	2.16783103
80%	587	2.05671254

Now, we will show a histogram that is focused on the execution time variation

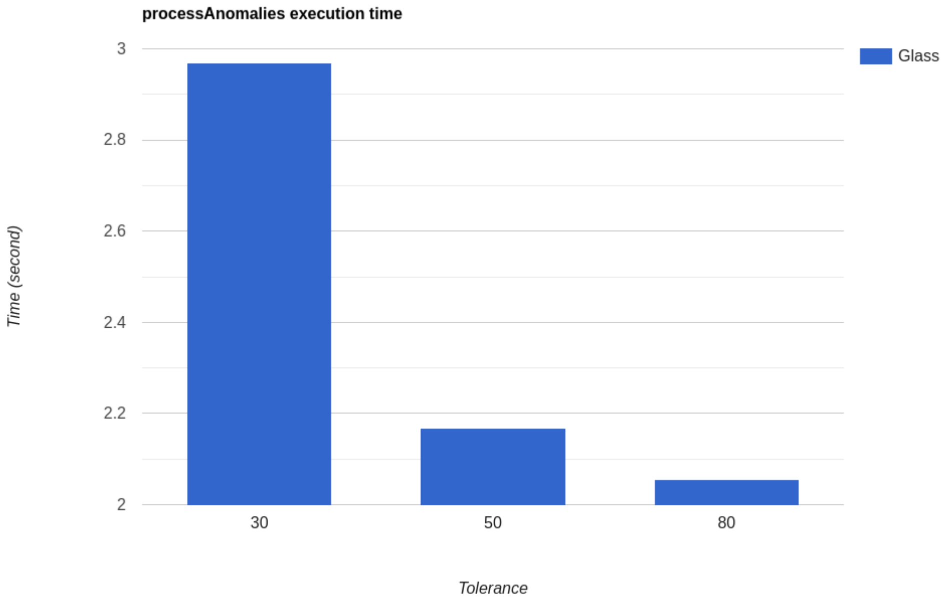


Figure 17: glass identification performances

5.2.2 wineQuality

The second dataset analyzed is wineQuality, which is a medium size and much bigger than the glass dataset. It's observable how the time is increasing. With this dataset, we did 50 measurements, obtaining this information:

tolerance	**anomalies**	**average time(s)**
30%	5095	15.18440648
50%	2995	12.1148551
80%	1319	9.78395132

Now, we will show a histogram that is focused on the execution time variation

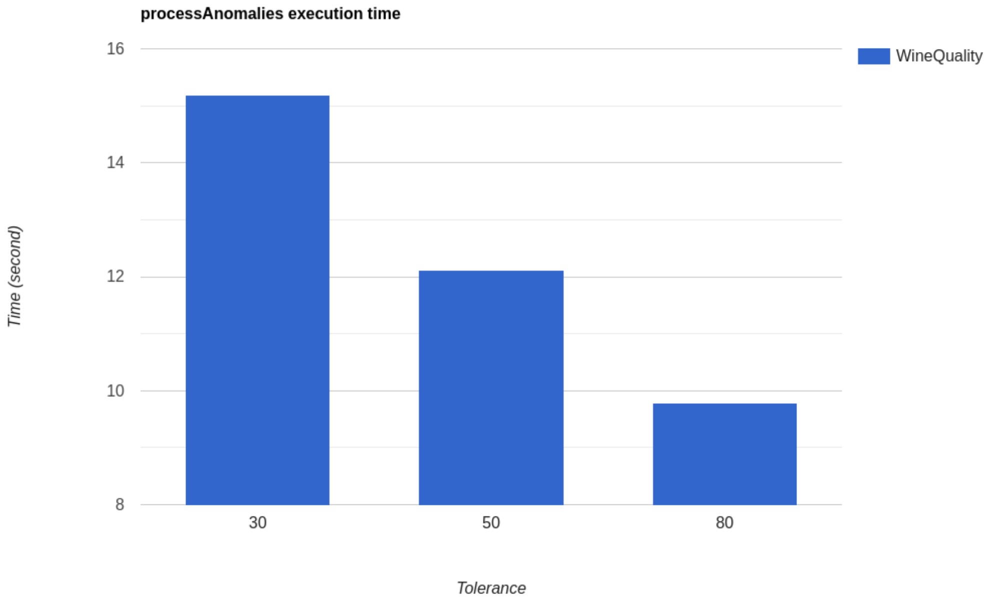


Figure 18: wine quality performances

Differently from the previously analyzed dataset, here the time increase is regular, adding approximately 2, 2-3 seconds when the tolerance percentage decreases.

5.2.3 airQuality

The last dataset analyzed is also the largest one for entry number and fields number. Obviously, that determines a stressful test for the software. This is the result of the analysis:

tolerance	anomalies	average time(s)
30%	93820	74.16534109
50%	74545	61.15170070
80%	58921	53.39762475

Like the other datasets, also for this one, there is a histogram that shows how the execution time changes:

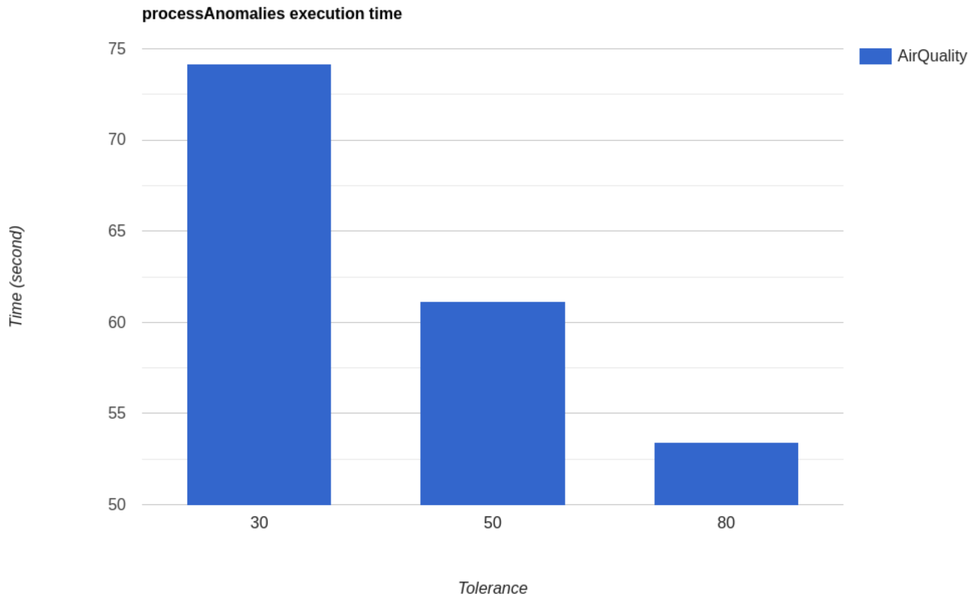
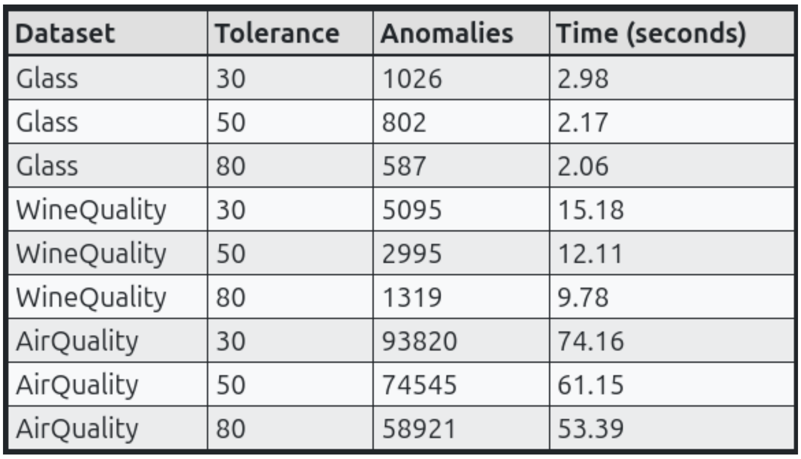


Figure 19: air quality performances

5.2.4 Comparison between datasets

Now, we can observe the results obtained compared together:



Dataset	Tolerance	Anomalies	Time (seconds)
Glass	30	1026	2.98
Glass	50	802	2.17
Glass	80	587	2.06
WineQuality	30	5095	15.18
WineQuality	50	2995	12.11
WineQuality	80	1319	9.78
AirQuality	30	93820	74.16
AirQuality	50	74545	61.15
AirQuality	80	58921	53.39

Figure 20: Compared Datasets Table

As we can expect, the higher number of entries, combined with the lower tolerance percentage, increases the execution time exponentially

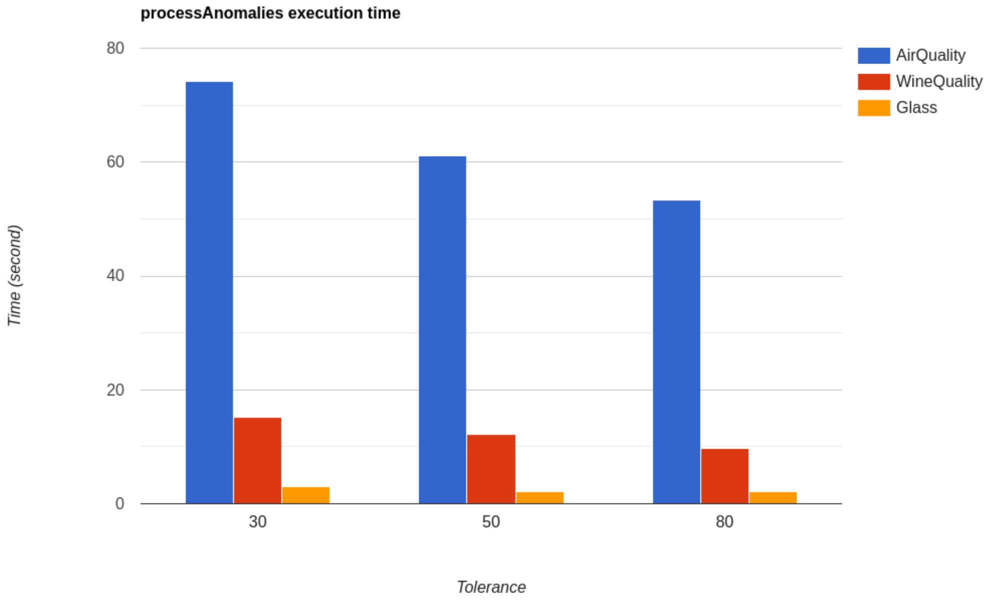
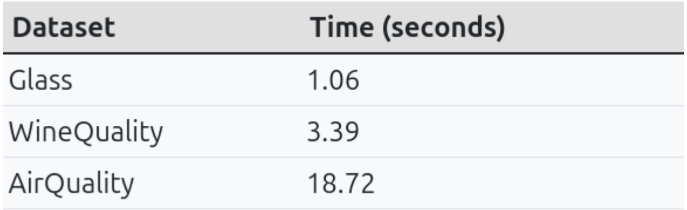


Figure 21: Compared Datasets Histogram

5.2.5 Performance Analysis with infinite tolerance

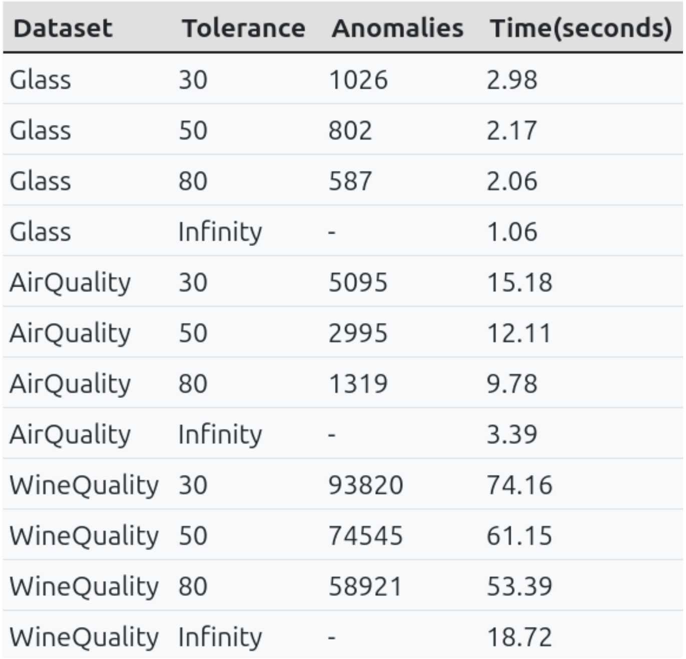
The idea is to show the effective time spent by the function to perform the check analysis, without the time used to perform the insert in the database. So, applying an infinite tolerance, the software doesn't find any anomaly and we can measure the effective time of the execution



Dataset	Time (seconds)
Glass	1.06
WineQuality	3.39
AirQuality	18.72

Figure 22: Compared Datasets No Insert

In the end, we want to show the final comparison between the dataset with the last measurement done with infinite tolerance



Dataset	Tolerance	Anomalies	Time(seconds)
Glass	30	1026	2.98
Glass	50	802	2.17
Glass	80	587	2.06
Glass	Infinity	-	1.06
AirQuality	30	5095	15.18
AirQuality	50	2995	12.11
AirQuality	80	1319	9.78
AirQuality	Infinity	-	3.39
WineQuality	30	93820	74.16
WineQuality	50	74545	61.15
WineQuality	80	58921	53.39
WineQuality	Infinity	-	18.72

Figure 23: Final Performance