National Research University Higher School of Economics

Faculty of Business and Management

Department of Innovation and Business in Information Technologies

Master thesis

Data Anomalies Investigations in High-Load Systems

Student Korobkov Anton

Group MBD151

Supervisor Yevgeni Koucheryavy

2017-05-21

Moscow 2017

Contents

1 Introduction

1.1 Data anomalies in modern information systems

Detecting anomalies in data is a complex and error-prone process that usually requires human participation. If the dataset is complicated enough, and stream of data is continuous and sufficiently large, manual identification of outliers becomes impossible. Consider a following dataset full digital trace of Internet user, for instance what web sites user visits, what web application she uses and what type of advertising she consumes. Detailed analysis of such dataset might be extremely valuable. In order to create high quality analytics based on this data, certain problems must be solved first:

1. How to detect changes in user preferences? For instance, how we can distinguish seasonal change of data (users tend to surf the Internet more during evenings and weekends) and real change of preferences, when certain Web resource becomes significantly more/less popular?

2. How to monitor specific unhandled cases in parsed data? New types of web content, consumed by users, are introduced on a regular basis, so there is a need to adapt the system to track them. It is too expensive and inefficient to introduce specially training engineer, so this process should be automated.

3. How to implement monitoring system that will be scalable (be able to query typical data storage based on NoSQL database), extendible to support decision making process and maintainable.

   Let us start from the most important part - data anomalies. The most broad definition of data anomaly is a data point (or pattern) that does not conform to expected behaviour. For instance, both unexpected declining trend in data and extremely high value are considered anomalous. There are various areas where finding anomalies in data are crucial: for instance, finding tumours[6] or detecting fraud in financial systems[5].

Finding anomalies in data is well-studied subject of statistics, with early application arose as early as 19th century[11]. Different anomaly detection techniques have been developed in several research communities over time. Many of these techniques have been specifically developed for certain application domains, while others are more generic[4]. In our work, we, however, mainly consider a problem of building such system. The rest of the work is organized as follows: we first perform literature review to understand existing approach to our problem, than we introduce the Beholder - a system to monitor heterogenous distributed high load system, its purpose and architecture, briefly describe main data sources. After that, we outline algorithms that we used to identify anomalies. The next section is devoted to system implementation and integrating it into business processes.

1.2 Literature review

First of all, it must be stated, that anomaly detection is somewhat narrow section of a more broad topic - root cause analysis. So we, in order to build reliable system, must try to investigate different approaches that already exists within this field.

First of all, there is a growing demand for systems that operates with huge volumes of data. The SMAC industry (Social, Mobile, Analytics and Cloud) Software as a Service and the Internet of Things more organizations in all industry sectors recognize they are evolving into technology and data companies[3]. It means that currently we are facing a transition of entire industries through software. Fast-paced growth of clouds and distributed it products poses serious challenges to IT leaders. When one is developing[1] somewhat complicated products it is very challenging, because it requires coordination of network connectivity, application protocols, data analytics and system management. In particular, in order to control any "Big Data" system, a throughout monitoring is required. Root cause analysis as a field of computer science is capable of precisely synthesizing the status of the system for human beings to make decisions. Specifically, human beings will no longer be capable of controlling so complex system through traditional dashboards and will require a higher level of automation to generate hypotheses of potential root-causes much more accurately[2]. While several decades of research have produced a large number of algorithms and techniques to perform root cause analysis in various fields, there is still lack of understanding of how to apply it to Big Data systems. Particularly, the appropriate interpretation and management of the vast amount of data generated by these systems needs to be underpinned in order for them to become genuinely viable. In this review we will focus mainly on different RCA models, paying special attention to its applicability for our case.

Let us begin by introducing main definitions. In this review we will inherit ter-

[1] And then deploy and support

minology defined in [1], because this paper provides good overview of existing approaches. A comprehensive list of definitions should include follows:

(a) **Event** We call an event an exceptional condition occurring in the operation of a system.

(b) **Fault** Is an event that produce events, but is not produced by other events. In other words, it is the cause, a failure of a system

(c) **Error** Is a state of system when its state differs from its 'good' condition.

(d) **Symptom** A symptom is an visibly demonstration of faults. It might be both fault itself as well as external alarms.

Our main aim in this work is to detect syptoms that were produced directly by faults. Let us now observe main techiques that exist in this field. First of all, there are kernel based techniques. Kernel techniques consider entire sequence as a single element of analysis, and usually apply some proxy point technique by defining similarity kernel for sequence. Window based techniques are using more traditional statistics approach, observing a subsequence of points, a sequence or a window, and then apply some tests to calculate "anomaly score". Techniques of this kind require a strict definition of what anomaly is due to nature of statistical test. More sophisticated approach is to use markovian techniques. These include using probabilistic models to evaluate each element in sequence.

Let us now briefly describe these classes of techniques. First of all, let us describe main kernel technique approaches. Entire sequence of points is treated as a single point, and then is utilized by traditional anomaly detection based algorithm. It should be briefly described how these algorithms work. First of all, a pairwise similarity matrix is computed for the training sequences in S. Next, the test sequence S_q is compared against the similarity matrix to obtain the anomaly score for S_q, using a point based anomaly detection algorithm[12]. However, several different variants of this algorithm were proposed.

There was proposed a clustering based technique in which the training sequences are first clustered into a fixed number of clusters using k-method algorithm[13]. The anomaly score for a test sequence is then computed as equal to the inverse of its similarity to its closest medoid. Also probabilistic clustering techniques that do not require an explicit similarity matrix to find clusters in training dataset were used for detecting anomalies as well. One example might be a mixture of probabilistic suffix trees, represented as a cluster[14]. It is also possible to do clustering by applying maximum entropy model[15].

As we stated above, in order to decide if the whole timeseries is anomalous, we need to compare it to some "ideal" timeseries, then decide if it is really deviate significantly. A simpliest way to achieve so is to compute Simple Matching Coefficient (SMC)[16], where we count number of positions in which sequences are equal (produce a match). This algorithm is rather fast (linear complexity), but requires sequences of equal lenghth. Slightly modified approach involves computing a coefficient of similarity between two sequences () might be computed as follows:

$$nLCS(S_i, S_j) = \frac{|LCS(S_i, S_j)|}{\sqrt{|S_i||S_j|}}$$

where $LCS(S_i, S_j)$ is longest common substring (in this case - subsequence) according to [12]. This method has obvious disadvantages, which is computational complexity. Another method is proposed by[17] is covering the sequences into bitmap signature and compare those signatures to determine similarity.

Let us now consider techniques that use window-based approach. The main idea is very simple - we extract fixed lenght window of arbitry size and then apply an algorithm to a sequence of data points. The applicability of window techniques is justified because of fundamental limitation of kernel based techniques. When we apply kernel based methods we compare a sequence with some ideal model to decide whether timeseries is actually anomalous. However, this process is computationally expensive and limited, albeit theoretically

correct from mathematical point of view. So it is easier to localize a sample of data and apply more crude yet effective enough algorithm on such timeseries. Another important case for adopting techniques of this type is the fact that if outlier is a single datapoint in a series, we will find it occuring in more than one window, because windows as subsequence will overlap.

The classical technique is to slide fixed length window across sequence. Then some algorithm of choice is applied, let us consider an simple example called threshold based sequence time delay embedding[18], or tstide for short. Let us describe the steps of this algorithm. First of all, we set windows length, k and derive $N-k+1$ overlapping windows. Then, for each window ω_i we assign a likelihood score $L(\omega_i)$ which is equal to the frequency associated with the window ω_i in a so-called normal dictionary. We define normal dictionary as occurence of each unique window in the training data. Then a treshold λ is set to check if given window is anomalous. In other words, the anomaly score of the test sequence is proportional to the number of anomalous windows in the test sequence. It can also be expressed with a following formula:

$$A(S_q) = \frac{i : L(\omega_i) < \lambda, 1 \leq i \leq t}{t}$$

This method was modified multiple times, mainly to assign a different anomaly score to a window. We will provide sample of such techniques here. One way to improve this process is to use lookahead pairs to compute score of a window[19]. This objects are constructed as follows: For each symbol that occurs in S the symbols that occur j positions after that symbol $(\forall j \in [1,k])$ are recorded. To compute anomaly score for ω_i the number of symbol pairs of the form $(\omega_i[1], \omega_i[j]), \forall j \in (1,k]$ are counted such that $\omega_i[1]$ is not followed by $\omega_i[j]$ after j positions in the normal dictionary. The total number of unequal (mismatched) lookahed pairs is divided by k is the anomaly score for a window ω_i.

Another option is to compare against a normal dictionary. These techniques build normal dictionary as stated above, with fixed lenght windows, from training sequence and then compare each window taken from test sequence to the normal dictionary to obtain an anomaly score. One example might be an application proposed in [20] where authors used Hamming distance (difference of equal sequences) between test window ω_i and closest window in the normal dictionary as anomaly score $A(\omega_i)$. Authors also proposed different technique to compute $A(\omega_i)$:

$$A(\omega_i) = \left\{ \begin{array}{ll} 1 & \text{, if the window } \omega_i \text{ is not present in the normal dictionary} \\ 0 & \text{, otherwise} \end{array} \right\}$$

Another approach is proposed in [21] is how to compute different similarity $Sim(\omega_i, O_j)$ measure between a test window ω_i and a window, in the normal dictionary, O_j

$$w(\omega_i, O_j, l) = \left\{ \begin{array}{ll} 0 & \text{if } i = 0 \text{ or } \omega_i[l] \end{array} \right\}$$

The overall similarity is computed as follows:

$$Sim(\omega_i, O_j) = \sum_{l=0}^{k} w(\omega_i, o_j, l)$$

This, however, has one obvious disadvantage, mainly that one mismatch in a middle of a large sequence can result in great similarity reduction.

Fundamentally different approach is introduced in [22]. In this paper authors are applying hidden markov chains to solve classification problem. Firstly, a training is performed, which involves learning a classifier from the training data set S. If S contains both normal and abnormal sequences, it is possible to obtain labeled windows in a following manner: if window is extracted from normal

sequences, it is labelled as normal, and it is labelled as anomalous otherwise. It is possible for S to contain no anomalous sequences, in this case authors suggest generating it.

After the assignation of anomality score for each window, we have to combine it (a vector of length $|S_q| - k + 1$) to find an overall anomaly score $A(S_q)$, which is used to determine anomalous points.

Another class of anomaly mining technique is markovian methods. Markovian methods use approximation of the correct distribution, that generated the data[12]. In most cases, the probability of a Sequence S might be represented using chain rule:

$$P(S) = \prod_{i=1}^{l} P(s_i | s_1, s_2, ...s_{i-1})$$

where l is the length of the sequences, which is observed across multiple different domains[23]. It utilizes basic property of a higher-order Markov condition which states that the conditional probability of occurrence of a symbol s_i given the sequence observed so far can be approximated as[12]

$$P(s_i | s_1 s_2 ... s_{i-1}) = P(s_i | s_{i-k} s_{i-k-1} ... s_{i-1}),\ k > 1$$

Techniques of this kind operate in two phases, training and testing. A probabilistic model is build using training (this mechanism is similar to the one explained in "window based techniques"). The testing phase consists of calculating the conditional probability for each symbol of the test sequence, using the model, and then combining them to obtain an overall probability for the the test sequence. The overall anomaly $A(S_q)$ score is for the test sequence equals to inverse of the probability of S_q.

To summarize what we have just reviewed it might be sad that of these techniques we will apply windows-based approach (see section 2.2), because it is

most practical to extract a fixed number of recent data points and only after that apply algorithms of choice. There are two main reason to chose this type of approach: first of all, computation efficiency. We are analysing behaviour of a high load system, so it is an issue there, a data is in form of metrics (time series), which dynamically formed in large numbers (possibly more than 20000 new metrics daily), so it is not justifiable to use $O(n^2)$ algorithms here. In next part we will describe our approach to design a system that extracts data, applies algorithms and produce valuable insight on system misbehaviour.

2 Main part

2.1 Project "Beholder": specifications

In this section we describe how what system we are building and how are we going to implement it. Before we start developing it, we must derive technical specifications. As we stated earlier[2] we are going to build system that will perform analytical monitoring, to check in real-time (or nearly real-time) both the overall health of the system and health of distinct components. Following requirements for this system must be satisfied:

(a) Horizontal scalability. We need to be capable to plug in any number of various data sources we need. For instance, one of the core component of our main system is distributed MongoDB database, so we need to be able to query any number of MongoDB nodes at any given moment. It is also desirable to allocate computational resources in a manner that will allow us to do most of the heaviest computing on client side, because a machine Beholder will be deployed on is relatively weak in comparison to overall data processing system.

(b) Stability. Because we have multiple data sources and we are tracking data anomalies, we need to be able to keep the system up and running without any breakdowns, because in that case we will have false alarm that something is wrong with main system, but in reality that might be a failure of monitoring system. We will elaborate on this more in 2.3.

(c) One specialized data source to save aggregated data from different components of the system. Because we have many data sources to pull data from, we need to collect extracted data in one system that is capable of saving time series and has user friendly API to query and perform ad-hoc analysis.

[2] Will be elaborated in introduction

(d) Algorithms. We need to have easy to use, extensible modules that will allow us to apply chosen algorithms on specific problems with context

(e) Reporting system. We need to automate different kinds of reports for ops team, development team and non-technical users and provide those who will need custom data from our storage with all necessary access rights.

To sum up we need to build a service that will allow us to collect data, store data and then perform analysis (mostly ad-hoc). The amount of components to monitor and complexity of system led us to creation of dedicated service, which was called "The Beholder".

2.2 Data sources and project architecture

In this section we will describe the problems that we solved while designing this system. First of all, we will describe what is analytical monitoring, how can we efficiently perform it in general and then will describe how to apply it in our specific case.

We shall begin by describing the ecosystem we are operating in. We assume that our initial set up is that we have multiple different data sources. We also assume that some of our data source can be considered "Big Data" storages, and we show in our example how to set system to collect data from large scale distributed system (more than 6 TB daily).

For this problem, all data sources can be split into two different categories: native and non-native. First class of data sources is called native, because we can use native driver, like JDBC, to execute queries seamlessly. Non-native data sources are those which use more general protocols (like HTTP or TCP/IP), with notable examples like Hadoop or MongoDB. Let us elaborate further on why we have chosen a classification rule like that.This decision is based on a way to communicate and transfer data between the Beholder and main data sources.

For a case with non-native database driver, usually more abstract programming language libraries are used, while for native (in most case relational) database we can simply pass a query to engine for execution. This greatly affects how exactly we are designing database interaction. For a native data source it is very handy to use design pattern command[7] because we can easily encapsulate query as an independent object. We can set create our own API to standardize queries evocation - exactly what should be the format of output data, and then write all queries so they will return data in a similar format.

For non-native drivers, on the other hand, we need to use pattern builder[8], implemented as a query builder. Instead of creating independent objects, we will create multiple methods within single "command" object, and decorate

them in a way that is in line with API we manufactured. Performing this in a following manner gives us two main advantages:

(a) For most of the non-native data sources it is not very easy to build seamless query-like object, so query builder gives us more flexibility in terms of data manipulation. The downside of this, of course is the fact that we have to do more client-side computation, which in our specific case is not desirable (see more in 2.3)

(b) On the other hand, for native data source (which, in our case, are mainly relational databases) we can easily write maintainable, testable and easy to read queries that will allow us to extract any info we need

More detailed illustration below

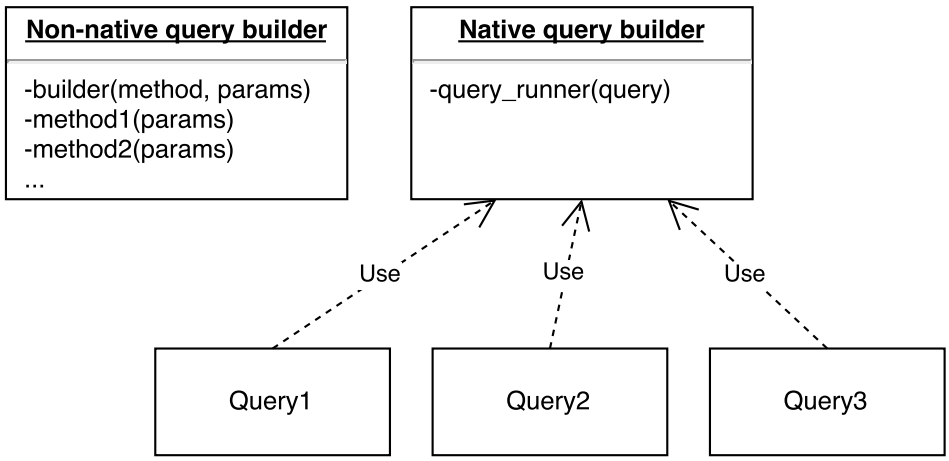


Figure 1: Queries architecture

Both native and non-native query builders should be initialized on runtime with all specific parameters set in order to run data collectors - the components that will do the exact data collection[3].

After we have discussed the possibilities of extraction of necessary data from different kinds of sources, we may now discuss what exactly we are going to extract. As we stated earlier, our point is to monitor overall health of the

[3]We will elaborate on implementation later in 2.3

system. So we need to track metrics that represent the state of different components of the system. We define metric as a time series that represent some aspect of the system quantitatively. So from each data source we aim to extract all the metrics that will be necessary for our analysis.

Metrics come in all shapes and sizes, but we can split them approximately in following groups.

(a) System metric. This includes information like CPU load, memory, monitoring state of virtual containers, etc. This data is not of a particular interest to us, because we are building analytical monitoring and usually there is an ops team that in charge checking this.

(b) Data completeness. Due to the nature of system Beholder is built to monitor (and for many similar cases with significant data processing), we are in need of checking that certain type of data has arrived in a certain place. For instance, a specific check if all files are loaded from storage system into database.

(c) Quantitative measure to check specific parameter. The most interesting case, where we need to grasp a nature of specific event that occurs at a given time.

Obviously in this work we are more interested in a third type of metrics, so we will discuss how to deal with it later on. It should be noted that monitoring metrics of a second type is trivial, so we will not shed light on details here.

An architect of a system usually can pick storage technologies to suit business needs, but in our case we can only choose the engine to store metrics. It basically means, that we have to deal with whatever storage engine project is using in order to extract data as metrics. In order to fully understand how to build a system to extract metrics from, let us briefly discuss how different data base storages are convenient to do so.

Let us start from the most basic one - relational databases. It is also the only

native (see the definition above) database system we will review here. First of all, it should be noted, that it is not necessary to follow architecture we defined, because it is possible to apply pattern query builder here as well - there exists various ORM[4] libraries that allow its user to create database agnostic interaction, but for simplicity we will not use them and will continue to apply pattern "command" in our case. Another reason not to use ORM is that it might be extremely painful and complicated to write a sophisticated query, also one of the most important feature of a relational database is a `join` operator, and joining tables with ORM tend to produce unpredictable memory leaks. But if we use encapsulated commands to query specific databases, then extracting time series metrics becomes relatively easy process. There are few important points, however, that system architect should take into consideration. First of all, when one is collecting a metric, she, in most cases, has to perform a query optimization[5]. This is important because monitoring system is actually a peripheral service and we need not to put computational burden on a production service. Another important point is that we have to carefully design API so that all queries will return time series like so:

datetime	metric_name	value
2017-01-10 00:00:00	eywa.video_ad.computed	1321
2017-01-10 00:00:00	eywa.video_ad.viewed	672

The next popular type of database is document oriented databases. We will discuss in details how to extract metrics from this type of databases due to the fact that this is a very popular choice to store log data or data with many nested levels and these kind of data usually needs through monitoring, because logs

[4]Object-relational mapping - a technique to cover native objects of a programming language of a choice to objects that can be used to communicate with database

[5]This is indeed the case not only for relational databases, but for any kind of storage engines, but for rdbs it should be specially highlighted due to the complicated (in most cases) database schema and sheer amount of logical intersection

usually contains error and warning messages, and complicated document structure in many cases may be due to extensive parsing of different objects, which is prone to heisenbugs. Most of document oriented systems are distributed so we need to extract document hashes first in order to identify to what server specific document belongs. Then we have to prepare specific query builder. It also worth mentioning that in most cases document-oriented database means that we can fully leverage map reduce paradigm.

Another important type is perhaps the most tedious one - raw data files. The most notable example here is Apache Hadoop, which essentially a collection of documents, scattered across nodes. The process of extracting data from such storages might get really tedious as we have to write different handlers for different data format. In case of Hadoop we can use clients still, but, for instance for remote log parsing we have to write dedicated handlers. However, even this case is well aligned with our usage of builder pattern.

Perhaps the most interesting type of databases when one is dealing with collecting and using metrics is time-series database. Time series database is a software system that is optimized for handling time series data, arrays of numbers indexed by time (a datetime or a datetime range). It means that it fits the needs of our project very well, and we will later show how to use it to store metrics for later usage, but they also quite capable of being a source of data to extract. In most cases the interaction between database of this kind and client software is performed via API, so query builder will do.

After we briefly observed means to collect metrics and basic data sources, we may proceed further to designing analytical pipeline. The idea here is that we use all the data we have collected in an aggregated form, and launch a series of periodical checks to determine if the main system is not malfunctioning in any way. The problem here is that system anomaly is a very broad category to define. We can contingently split this kind of data into following categories:

(a) An error. In most cases, in error is manifested as sudden absence of metric

value, or, rarely, as sudden appearence of some variable. Usualy this kind of anomaly is caused not by changes in source data the system is working with, but by complete failure of one or more components. It is also might be caused by failure of monitoring system, and that is the reason why the whole system should be covered by tests, so the number of false alarms produced by low quality software will be minimized.

(b) An outlier. The most "classical" anomaly - sudden rise or fall in data. It is important to take into account seasonality in data to understand whether particular point is an anomaly.

(c) Continious trend. It is a case, which may be extremely tricky to locate even for a human observer, when we have time series rising or declining over time. There are certain techniques to locate such an heteroscedasticity test.

It is logical to assume that next step here will be discussing algorithms. Let us also consider anomalies of a second type, e.g. First of all, let us consider a case where we do not have hourly seasonal data. We will derive following algorithms (written in pseudocode for simplicity).

First of all, slightly modified Grubbs criteria:

Data: ts time_series_data
Result: Is anomalous bool
initialization;
mean = avg(ts);
stdev = stdev(ts);
tail = ts[-3::];
▷ We selected three last points of time series to increase their weight
z_statistics = (avg(tail) - mean) / stdev;
anomaly_threshold = rev_surv(0.05 / 2 * len(ts), len(ts) - 2);
threshold_squared = anomaly_threshold ** 2;
grubbs_score = (len(ts) - 1) / sqrt(len(ts)) * sqrt(threshold_squared / (len(ts) - 2 + threshold_squared));
if *z_statistics* > *grubbs_score* **then**
    return True;
else
    return False;
end

Whrere rev_surv is reverse survival function, implemented in Scipy Python package[9].

Second algorithm is analysing absolute deviation from median. We assume that a timeseries is anomalous if the deviation of its latest datapoint with respect to

the median is X times larger than the median of deviations.

Data: ts time_series_data, stat int

Result: Is anomalous bool

initialization;

median = median(ts);

demeniaded = array([]);

for *val in ts* **do**
    d

end

emeniaded.append(abs(val - median));

median_deviation = median(demeniaded) **if** *median_deviation == 0* **then**

    return False;

else
    t

end

est_statistics = demeniaded[-1] / median_deviation;

if *test_statistics* > *stat* **then**

    return True;

                                ▷ It is found heuristically that the best stat is 6

else

    return False;

end

Next criteria is that a timeseries is anomalous if the average of the last three

datapoints falls into a histogram bin with less than 20 other datapoints[6]

```
Data: ts time_series_data, bins integer, binsize integer
Result: is_anomalous bool
initialization;
tail_avg = avg(ts[-3::]);
hist = histogram(ts, bins);
bin = hist[1];
for index, bin_size in bin do
    if binsize <= 20 then
        if index == 0 then
            if tail_avg <= bin[0] then
                return True;
            end
        end
        if tail_avg >= bin[index] and tail_avg < bin[index + 1] then
            return True;
        end
    end
end
return False;
```

We also applied number of more trivial algorithms. We assume that a time-series is anomalous if the absolute value of the average of the latest three data point minus the moving average is greater than three standard deviations of the average. This does not exponentially weight the MA and so is better for detecting anomalies with respect to the entire series.We also assumed that a time-series might be considered anomalous if the value of the next data point is farther than three standard deviations out in cumulative terms after subtracting the mean from each data point. We also consider time-series anomalous if if the absolute value of the average of latest three data points minus the moving

[6]Arbitrary number, tweak with respect to data

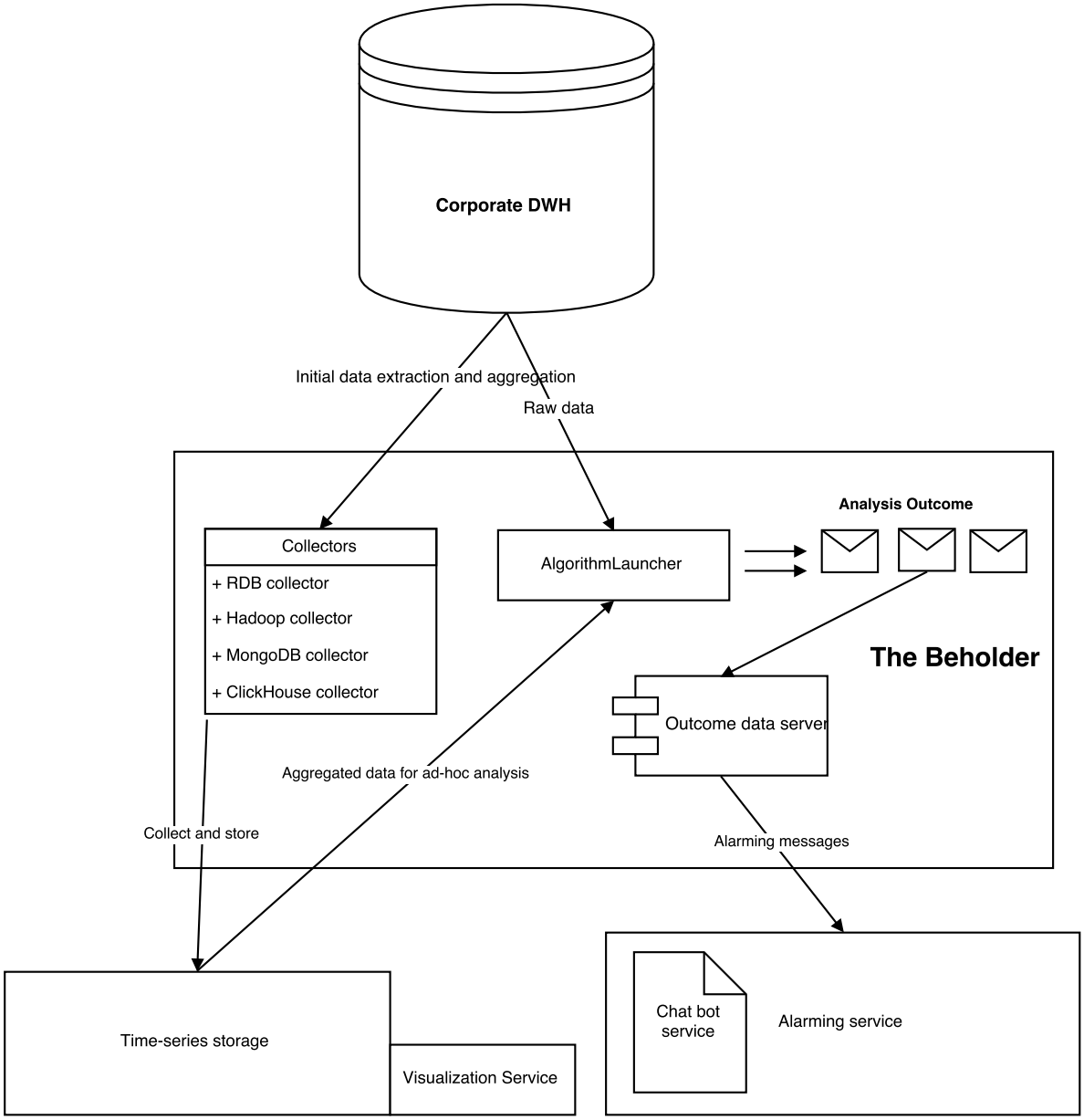
average is greater than three standard deviations of the moving average. This is better for finding anomalies with respect to the short term trends. Another criteria is that a time-series is anomalous if the value of the next data point in the series is farther than three standard deviations out in cumulative terms after subtracting the mean from each data point. In order to locate all true positive we propose a following ensemble technique: we compute the number of criteria that yielded "True" and if it surpass certain arbitrary threshold, we conclude that this time series is anomalous (see details in section 2.4)

These algorithms can be easily used when we are looking for outliers in data without seasonality, but when we are checking data with seasonality we need different approach, because we will have too much false positives triggered otherwise. Searching for anomalies in data with significant seasonality component is very challenging task and deserve many papers on its own, so we decided to limit our work in this area and we just applied basic algorithm[10], implemented by Twitter and available as a library for R programming language.

Now that we defined the algorithms, we need to design alerting system. All checks are useless, unless their results are not passed to decision maker, so this component of the system is in fact crucial. First of all it should be stated how exactly we call algorithms to preform checks. The details will be provided in 2.3 and for now we just state that we are running scripts that perform checking via cron. We build a service to save data in form of JSON documents in order to make it reusable and accessible for different alarming components. By "different alarming components" here we mean that we will use both standardized way of alarming (sending alarms via mail) along with more unconventional ones, for instance, chat bot for messenger.

We should also briefly mention data visualization. As we stated early, we store data in a specific time-series database. The consequences of this is that we do not need to "reinvent the wheel" and create our own solutions, for we can have all that we need right out of the box.

Now we can combine all the components and draw them as a scheme.


Corporate DWH
Initial data extraction and aggregation
Raw data
Analysis Outcome
Collectors
+ RDB collector
+ Hadoop collector
+ MongoDB collector
+ ClickHouse collector
AlgorithmLauncher
The Beholder
Outcome data server
Aggregated data for ad-hoc analysis
Collect and store
Alarming messages
Time-series storage
Visualization Service
Chat bot service
Alarming service

This architecture binds together individual modules we described above. We have a dedicated server, that communicates with main DWH, which, in fact, not a monolithic machine but many data storage services, and then apply algorithms on data we extracted earlier. The server then sends data to whatever module is reading. Let us now describe how exactly we have implemented it.

2.3 System implementation

Before we describe the results of system implementation, we need to set up a framework of subject area. The main product (in our context a system that we were monitoring) is the system that includes several components which are installed on users computer or another device with internet access (mainly browser extension). It monitor all user's traffic in order to filter out how much content user is consuming. This system is relatively complex in term of business logic: e.g. it is capable of understanding exactly what user is doing, if she reads the blog or watch advertisement video. There also is a subsystem that had been built with a cooperation with most of the biggest runet web platforms, which collecting data on daily visits (total amount of hits on different web site section). The problem here is that the Internet is very liquid and changing system, and therefore the structure of data that we produce and sell changes on a regular basis. These datasets contain high business value because agents, who deliver target advertising are basing their system on reports produced by it. So the idea was the following: we deploy entire set of collectors, then schedule their launch, then construct a business process around it.

Let us begin with the implementation of architecture that was stated in previous section. First of all, accessing corporate DWH in order to extract time-series. We created adapters to various RDBs and NoSQL solutions as well. The most challenging aspect here is the fact that most recent data for different time series is updated with different lags and may be hourly, daily, weekly, or even be collected on a quarter-hour basis. Using terminology we stated early, the main point is to locate **events** that were directly caused by **faults** and **events** that were not the product of system misbehaving, but rather an outcomes of interesting changes in data, for instance trends. In other words, there are cases when reliability engineer should be informed in a matter of minutes, and cases when no actions should be taken but decision makers should be informed about changes in data. Because of these reasons we cannot really create unified "start-

ing point" and have to set manually lag values for sets of metrics.

The next, and the most important component, is Beholder server itself. We decided to implement Collectors are simply modules that imported from so-called launching scripts. Because we used modern virtualization techniques[24], we packed all the environment into single virtual container to simplify redistribution of the software and to make launching script runnable from host system. Then, using any task scheduler (usually cron) it is possible to run everything on time from host system. It is also reasonable to run algorithm launcher from the same scheduler as data collector and with similar periodicity, because the nature of the process is similar. We decided to store algorithm output in a form of .JSON files, because it is easiest form to parse. Beholder also includes dedicated service application that serves purpose of serving those files. The idea is that any third-party service should be able to read outcome of the analysis to be informed. Currently, it is communicating with analytical chat bot and Zabbix monitoring solution, maintained by ops team.

Let us discuss the next component, chat bot alarming service. Using chat bots in analytical systems is somewhat novel approach[27] but it turned to be rather fruitful. A messenger called Telegram provides convenient development environment to observe behaviour of the remote system. There might be a problem with integrating chat bot with data source, but is is mere a technical obstacle. The chat bot in its final form worked as follows: first it would take a signal in a form of command from user, and then it would periodically fetch data from beholder and alarm the user if something is wrong. A slight modification however is applied to "cache" events that are happening multiple times; in other words, if something goes wrong, bot should tell user only once

Perhaps the most interesting and challenging here is building time series storage correctly. The problem is, we have to store hundreds of thousands of metrics and, according to our architecture, extract data on the fly and analyse it quickly. We will briefly observe main solutions here and describe why we picked exactly this option. First of all, we need an option to create met-

rics dynamically. We also need to communicate with API, preferably not very complicated and computationally efficient, because the system have to scale well. Ideally, we would like to have horizontal scalability as well. And, finally, before everything automated, we need a convenient way to observe time series in graphical representation. We decided to pick Graphite as a metric back end, Grafana as an instrument to navigate through data plots and influxDB to store specific time series. Let us describe this in details.

First of all, it should be said about graphite. It is an enterprise-ready monitoring tool that is capable of storing, and rendering time series in a form of dynamically allocated trees[26]. It is made of three different elements (web application, dedicated storage and metric gatherer), which is an obvious disadvantage, but is very simple to set up and maintain, has very well documented and user friendly HTTP API. On the other hand, it has no horizontal scalability and relatively slow in general. But comparing to other monitoring solutions, which are Prometheus and InfluxDB, which do not scale as well, it provides the easiest way to store data once loaded.

The next component is visualization service. As we stated earlier, we need specific tool to plot time series. One option would be to create self-made visualization tools using various libraries available out there (like matplotlib for Python). But the better option is to apply prebuild solution. 1 decided to deploy and use grafana. Grafana is the open platform for analytics and monitoring, which allows its user to build any kind visualisation she wants. It is in fact an information system, not data storage, so it requires back end to operate. It supports all the popular time series databases, including our choice (graphite and InfluxDB).

And finally, specific storage for specific cases. Graphite, in its simplicity, allows only to store one numerical value and assigned time stamp for each data point. However, in some cases, we need more than that - a functionality that will be similar to relational database, but with schema easier to create. Modern time series database InfluxDB provides exactly this, and also allows to create

so called fields and valuesInfluxDB that allow user to store different types of data (text or numerical respectively), to make column indexing easier.

2.4 From design to production

After we discussed all the necessary component of the system and elaborated on implementation details, we may now describe how this system was integrated into existing business process.

First of all, there is one notable difference between the part of the system that is responsible for monitoring user behaviour and the one that contains data on hits differences in terms of business processes. Monitoring user behaviour is in fact monitoring the software itself, managers are unaware of what is going on under the hood, while when hits on specific site breaks, it usually attracts attention of managers and analysts who work with those clients. So, in our case, we have two main users of these system:

(a) An ops team and developers. These users are mainly interested in knowing that something has gone wrong with a system that some specific **component** of the system they are responsible for

(b) Web analysts[7] and managers. These people indeed are not interested at all in what is going on deep within the system, but they need to know if something is wrong with distinct **client data**

The overall amount of information is too large for a single individual to process - as we stated earlier, we have hundreds of thousands of metrics that measure system performance. It makes it completely cost ineffective to assign specific employees to monitor these parameters. Another reason is that personnel start to ignore repetitive false alarms[8] and lose trust to the system. So it should be convenient, which means less false positives, and more "broad" checks. We also figured out that reliability engineers become more engaged into the whole process if we allow them to write modules that listens to the Beholder and process its data, after main algorithms are applied, with their own means themselves, but the number of observations in a sample was relatively small, so we

[7] In this context 'web' refers to analysts that solve specific client problems

[8] It might be refereed as "A boy who cried wolf" effect

can’t say if this anecdotal evidence bears any statistical significance. The first attempt was by sending detailed data directly into Zabbix (see section "System implementation") above, which led to over flooding engineers with data. It failed, because they simply piped all the output back to the developers of the monitoring system. The next attempt was more successful - we created one single state condition for the entire system - whether it is unhealthy or not. When engineers were provided with this, it has become very easy for them to operate, and this option was eventually released.

Managers, however, are more interested in symptoms. So it was necessary to provide them with details, but data should be compact and concise. We applied algorithms that we defined in section "Data sources and project architecture" in order to achieve this. We constructed an ensemble (see algorithms above) and, using Grafana dashboard, set a specific cases to alarm decision makers only if something goes wrong. This approach has proven to be fairly successful, as managers and analysts are able to track disturbances in data themselves and only after that report to developers or clients.

3 Conclusion

In this work we considered a problem of designing and implementing a system to monitor what essentially is any kind of disturbances in data. We have shown that we can tweak and tune existing algorithms to create comprehensive analytical monitoring system. We also present robust architecture and demonstrate how it should be implemented to fit into existing business processes. There is indeed a potential for studying this subject further. We can, for instance, increase overall performance of system by implementing caching mechanism on analytic algorithms invocation or do data streaming instead of sending batches. What is most important, however, is how to fit this system into whole business process. Alarming should be designed very specifically, otherwise it will not be possible to use insights gathered and processed by the system.Further research on how to build proper algorithms are definitely needed as well. Also we might consider more interesting algorithms to process obtained data with. So it might be said that more sophisticated engineering as well as theoretical approach might be applied to increase potential of such system as well.

References

[1] Kavulya S. P. et al. *Failure diagnosis of complex systems* Resilience assessment and evaluation of computing systems. - Springer Berlin Heidelberg, 2012. - . 239-261.

[2] Sole M. et al *Internet of things (iot): Survey on Models and Techniques for Root-Cause Analysis* arXiv preprint arXiv:1701.08546. – 2017.

[3] Gubbi J., Buyya R., Marusic S., and Palaniswami M. *Internet of things (iot): A vision, architectural elements, and future directions* vol. 29, no. 7, pp. 1645 - 1660, 2013.

[4] Chandola V., Banerjee A., Kumar V. *Anomaly detection: A survey* ACM computing surveys (CSUR). - 2009. - T. 41. - №. 3. - . 15.

[5] Aleskerov, E., Freisleben, B., and Rao, B. *Cardwatch: A neural network based database mining system for credit card fraud detection.* Proceedings of IEEE Computational Intelligence for Financial Engineering, 1997

[6] Parra, L. and Sajda, P. and Spence, C.D. *Detection, synthesis and compression in mammographic image analysis with a hierarchical image probability model.* Proceedings of the IEEE Workshop on Mathematical Methods in Biomedical Image Analysis, 2001

[7] Freeman, E., Robson, E., Bates, B., Sierra, K. *Head first design patterns* O'Reilly Media, Inc., 2014

[8] Gamma, E. *Design patterns: elements of reusable object-oriented software* Pearson Education India, 1995

[9] Jones, E., Oliphant, T., Peterson, P. *SciPy: Open source scientific tools for Python* 001–

[10] Rosner, B. *Percentage Points for a Generalized ESD Many-Outlier Procedure* Technometrics, 25(2), pp. 165-172., 1983

[11] Edgeworth F. Y. *On observations relating to several quantities* Hermathena. - 1887. - . 6. - №. 13. - . 279-285.

[12] Chandola V., Banerjee A., Kumar V. *Anomaly detection for discrete sequences: A survey* /IEEE Transactions on Knowledge and Data Engineering. - 2012. - . 24. - №. 5. - . 823-839

[13] Budalakoti S., Srivastava A. N., Otey M. E. *Anomaly detection and diagnosis algorithms for discrete symbol sequences with applications to airline safety* IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews). - 2009. - . 39. - №. 1. - . 101-113.

[14] Ron D., Singer Y., Tishby N. *Learning probabilistic automata with variable memory length* Proceedings of the seventh annual conference on Computational learning theory. - ACM, 1994. - . 35-46.

[15] Pavlov D. *Sequence modeling with mixtures of conditional maximum entropy distributions* Proceedings of the Third IEEE International Conference on Data Mining. Washington, DC, USA: IEEE Computer Society, 2003, p. 251.

[16] Smyth P. *Clustering sequences with hidden markov models* Advances in Neural Information Processing, vol. 9. MIT Press, 1997

[17] Kumar N., et al *Time-series bitmaps: a practical visualization tool for working with large time series databases* SDM, 2005

[18] Forrest S., Warrender, C., Pearlmutter B. *Detecting intrusions using system calls: Alternate data models* IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews). - 2009. - . 39. - №. 1. - . 101-113.

[19] Forrest S. et al. *A sense of self for unix processes* Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on. - IEEE, 1996. - . 120-128.

[20] Hofmeyr S.A., Forrest S., Somayaji A. *Intrusion detection using sequences of system calls* Journal of Computer Security, vol. 6, no. 3, pp. 151–180, 1998

[21] Lane T., Brodley C.E. *Temporal sequence learning and data reduction for anomaly detection* ACM Transactions on Information Systems and Security, vol. 2, no. 3, pp. 295–331, 1999

[22] Gao T., Ma H.Y., Yang, Y.H. *Hmms (hidden markov models) based on anomaly intrusion detection method* Proceedings of International Conference on Machine Learning and Cybernetics. IEEE Computer Society, 2002, pp. 381–385.

[23] Ron D., Singer Y., Tishby N. *The power of amnesia: learning probabilistic automata with variable memory length* Machine Learning, vol. 25, no. 2-3, pp. 117–149, 1996.

[24] Merkel D. *Docker: lightweight linux containers for consistent development and deployment* Linux Journal. - 2014. - . 2014. - №. 239. - . 2.

[25] Nezhad H. R. M. *Cognitive assistance at work* 2015 AAAI Fall Symposium Series. - 2015.

[26] The Graphite Project *http://graphite.readthedocs.io/*

[27] Nezhad H. R. M. *Cognitive assistance at work* 2015 AAAI Fall Symposium Series. - 2015.

[28] Leighton B. et al. *A best of both worlds approach to complex, efficient, time series data delivery* International Symposium on Environmental Software Systems. - Springer International Publishing, 2015. - . 371-379.