UDC 004.42 https://doi.org/10.20535.2786-8729.5.2024/316366

# SOFTWARE FOR COLLECTING AND ANALYZING METRICS IN HIGHLY LOADED APPLICATIONS BASED ON THE PROMETHEUS MONITORING SYSTEM

**Inna Stetsenko**
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
https://orcid.org/0000-0002-4601-0058

**Anton Myroniuk ***
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
https://orcid.org/0009-0003-7212-2816

*Corresponding author: antonmyronyuk@gmail.com

This paper emphasizes the importance of collecting metrics during application operation for early detection of potential problems. The undisputed leader in this area is the Prometheus monitoring system, which, combined with Grafana – a platform for visualizing collected data in numerous graphs – becomes an indispensable tool for programmers and site reliability engineers. However, the average value of a certain metric is often unrepresentative, because it does not reflect a comprehensive picture. Instead, collecting metrics in terms of various quantiles over a long period is useful to identify even single instabilities. Still, the use of standard tools in the Python ecosystem may require a lot of server resources and long preliminary analysis, which can be quite costly for businesses from a financial point of view. That is why the development of a new approach for collecting and analyzing metrics in highly loaded applications based on the Prometheus monitoring system is relevant.

The research aims to improve the efficiency of storing metrics across different quantiles, which will create additional opportunities for further analysis.

A review of existing approaches for calculating quantile values on large data sets was conducted. Their comparative characteristics in terms of speed and memory usage were also presented. The chosen method was adapted for use with the real-time data stream and implemented as a Python extension for the official Prometheus library. It opens up opportunities for comprehensive monitoring of highly loaded systems in terms of both server resource usage and the quantity and quality of collected useful data. This solution can be easily implemented on large projects requiring continuous tracking of various metrics to ensure stable and uninterrupted service operation.

**Key words:** metrics, quantile, highly loaded applications, Prometheus monitoring system, Python.

## 1. Introduction

Today, active digitalization of services is taking place against the backdrop of the rapid development of computer technologies. Every rapidly developing business that wants to increase its profits should have its website or application where users can find relevant information, place orders, contact support, etc. As a result, the need for operational stability reaches a new level, because business revenues directly depend on this. To identify potential problems early engineers are helped by collecting various metrics during the operation of applications. The undisputed leader in this area is the Prometheus monitoring system, which, combined with Grafana – a platform for visualizing collected data in numerous graphs, becomes an indispensable tool for programmers and site reliability engineers. However, the average value of a certain metric is often unrepresentative, because it does not reflect a comprehensive picture. Instead, collecting metrics in terms of various quantiles over a long period is useful to identify even single instabilities. However, using standard tools in the Python ecosystem may require a lot of server resources and long preliminary analysis, which can be quite costly for businesses from a financial point of view. That is why developing a new approach for collecting and analyzing metrics in highly loaded applications based on the Prometheus monitoring system is relevant.

## 2. Literature review and problem statement

Prometheus is an open-source project that provides a set of tools for monitoring and alerting systems. Since its inception, many companies and organizations have adopted Prometheus. Today, the project has a very active community of developers and users and has become an industry standard [1].

Prometheus stores data as a time series, meaning that metrics are stored with the timestamp of when they were recorded along with optional labels – key-value pairs. This database allows for large amounts of data to be stored thanks to special compression algorithms and, at the same time, be optimized for fast queries over arbitrary time intervals [2 – 3].

Metrics are numerical indicators of software properties and specifications, collected directly during its operation. Their essence may vary from application to application: for a web server it could be the duration of processing a request, for a database – the number of active or idle connections, for message brokers – the size of the queue, etc. Metrics play a crucial role in understanding whether the application is working as expected [4]. For example, by having an indicator of the number of requests to the API (Application Program Interface), the moments of peak activity can be determined. So, the number of operating servers may be preventively increased to handle all the corresponding load.

The Prometheus monitoring system consists of many components. The following elements are required for operation [1]:

– Prometheus-server that collects and stores metrics, and also provides an interface for accessing them through a specialized query language called PromQL;

– client libraries that are integrated directly into the application code and expose “client-server” (usually in a separate thread), “main server” will periodically (configurable, every 15 seconds by default) fetch data from it.

Optional components designed for programmers’ convenience include:

– push-gateway for recording metrics externally;

– alert-manager;

– metrics exporters for databases, storages, message brokers, numerous APIs, etc.;

– various visualization tools (Grafana is the most popular).

Prometheus client libraries implement the following types of metrics [5] (in fact, this is done only for the convenience of the end user, on the server they are all stored in the same data format – as time series):

– Counter;

– Gauge;

– Histogram;

– Summary.

Counter is a cumulative metric representing a single numerical value that increases monotonically. It is usually used to count the total number of requests processed, messages consumed, and application errors that occurred.

Gauge is a metric representing a single numerical value that can increase or decrease. It is typically used to measure the current memory usage of an application or the CPU utilization level. This metric may also be used for counting the number of concurrent requests a server is processing at a given time, or requests received waiting to be handled.

Before moving on to the characteristics of the Histogram and Summary metrics, it is necessary to understand the concept of quantiles. Quantile is one of the numerical characteristics of random variables used in mathematical statistics. It helps provide a brief characteristic of a certain data distribution. The value of the $\varphi$-quantile ($0 < \varphi < 1$) of a numerical series that consists of $N$ elements will be the actual value of the element with rank $\varphi N$. In other words, it is the same as the value of the element with position $\varphi N$ in this series, sorted in ascending order [6]. For example, the 0.9 quantile is such a number $x_\varphi$ that 90% of the values in the sequence are less than or equal to this value. The median of a numerical series is the value of the 0.5 quantile.

A histogram is a metric that forms a sample of observations and counts its numbers in predefined segments [7]. For query time, these can be the following segments: 0.01 s, 0.05 s, 0.1 s, 0.5 s, 1 s, 2 s, 5 s, 10 s. This allows extraction of the indicator values in arbitrary quantiles, performing aggregation on the server upon request. However, it should be remembered that the accuracy of the

metric measurement directly depends on the selected number and values of segments. In addition, the use of Prometheus server resources grows rapidly increasing the number of segments.

The Summary metric, like the Histogram, contains a sample of observations, such as request time, but, unlike it, performs aggregation of values directly on the client side. This requires specifying the desired quantiles for calculation before starting the indicators collection. For example, 0.5, 0.9, 0.99, obtaining the metric value in any other quantile will be impossible, which is a limitation of this method. Unlike the Histogram, there is no need to specify the segments of the metric values, which means it is not necessary to know the approximate response time before collecting indicators, it is enough to specify the desired quantiles, and real values of the metric can be found accurately directly during its collecting. This approach has another huge advantage – significant savings in server resources since all data will be immediately saved in aggregated form. The disadvantage of not being able to obtain the metric value in an arbitrary quantile that was not specified before collecting the metric does not seem critical. Specialists are typically interested in the median (0.5 quantile) and marginal (0.9 or 0.99 quantiles) indicators, whereas it is not so important to know what the exact value was in the 0.8 quantile.

Unfortunately, the standard Prometheus library for Python completely lacks support for calculating quantiles using the Summary metric. It is possible to collect only the sum of indicators and their total number. This can be used to obtain the average value of the metric (not even the median) by finding the quotient. However, it is generally not an informative indicator and does not reflect the full picture.

The practical recommendations section of the official Prometheus library documentation [5] suggests using Histogram in such cases. However, this will be quite expensive in computation resources for highly loaded systems, and will also limit the period during which graphs can be built in Grafana. In addition, the greatest difficulty in working with histograms is the need to conduct a preliminary analysis of each metric to select the correct segments for it, otherwise, the collected indicators will be unrepresentative.

To demonstrate the drawbacks of existing approaches, we will create a numerical series over a certain period as follows: $x_i = m + d$, where $m$ is the minute value at the moment of adding $x_i$ to the series, $m \in Z$, $m \in [0; 60)$; $d$ is a randomly generated real value to emulate certain series variability, $d \in R$, $d \in [0; 10)$. At each time point $t$, we calculate the median of the series values in the interval $[t - 10; t]$, i.e. starting 10 minutes before $t$. The constructed median graph is shown in Fig. 1. Here we can clearly see the dependence of the metric on the time point with some variability, which looks like a little waviness.


Fig. 1. The graph of the actual median of the series at 10-minute intervals

Let us construct the median (0.5 quantile) for this numerical series using the Histogram metric at the same intervals in 10 minutes. Assume the distribution of values is unknown in advance, therefore default segments will be used. The constructed graph is shown in Fig. 2. We see that it is very different from the original distribution in Fig. 1. We can only conclude that some of the values were less than 10 and the rest were greater but the full variability is not visible. This is an example of an incorrect selection of segments for the Histogram metric.

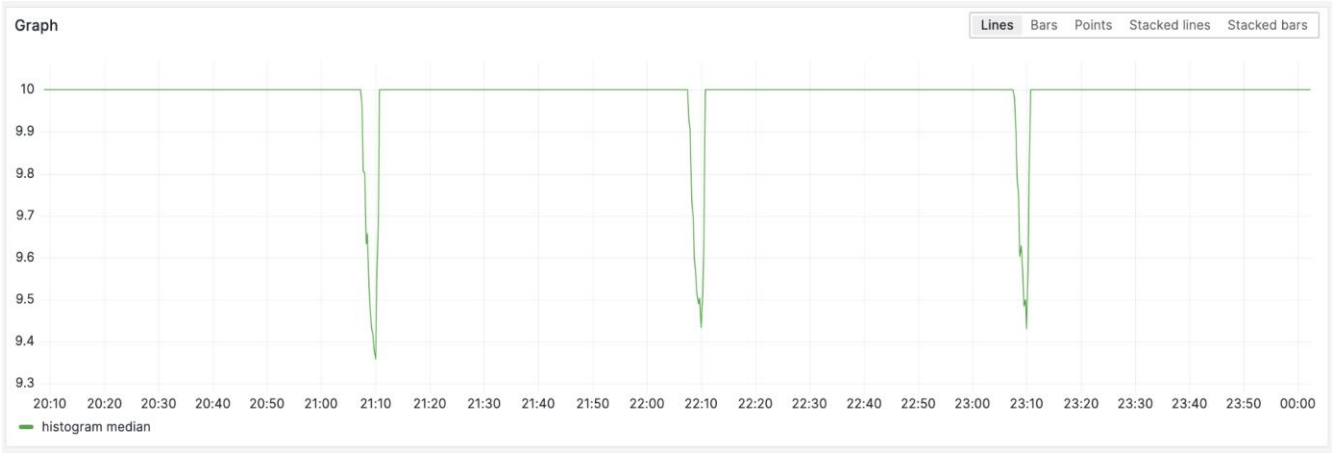


Fig. 2. The graph of the median of the series at 10-minute intervals, built with Histogram

There is an alternative implementation of the Prometheus client library for asynchronous applications called aioprometheus, which provides support for calculating quantiles for the Summary metric. However, it has a big problem – for obtaining the quantile value, the entire previously collected numerical series (starting from the moment the application was launched) is used. Because of this, over time, the metric will simply become a constant and will not show the real state of affairs at all, which is especially noticeable at the 0.9 and 0.99 quantiles in Fig. 3. In addition, the main process will experience a constant increase in RAM and CPU consumption for storing the full data set and calculating quantiles on it, which may cause unexpected problems in the application (force stopping due to SIGKILL, CPU throttling). Moreover, this library cannot be used for synchronous Python frameworks, such as Django and Flask, the main market leaders, so it cannot be called universal.

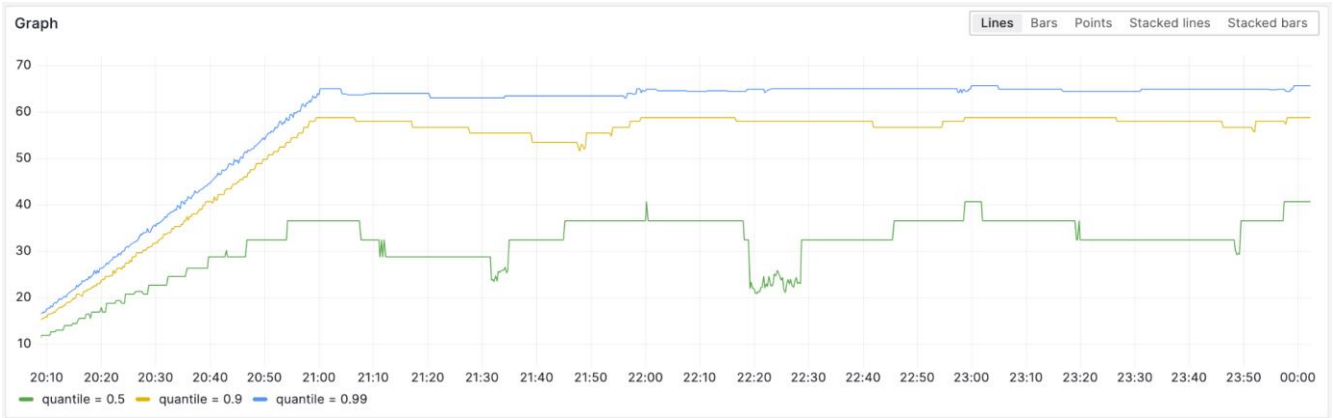


Fig. 3. The graph of series quantiles built with aioprometheus Summary

The existing approaches for collecting metrics across quantiles in the Python ecosystem either do not work properly or are too resource-intensive. So, there is a need to create a new solution that could be used efficiently in highly loaded systems to produce large amounts of various metrics.

## 3. The aim and objectives of the study

The research aims to create additional opportunities for collecting and analyzing metrics in highly loaded applications based on the Prometheus monitoring system by calculating quantiles on the client side.

To achieve the goal, the following tasks are set:

– to perform analysis of algorithms for calculating quantile values on the client side;

– to adapt the chosen algorithm for use with real-time data;

– to implement it as a Python extension for the official Prometheus library.

It should allow end-users to easily collect metrics across quantiles using Summary for synchronous or asynchronous applications with an arbitrary load level.

## 4. The study materials and methods of collecting metrics across quantiles

### 4.1. Quantiles calculation approaches

The simplest approach to calculating quantiles is to keep the complete data set. By definition, the quantile value is the value of the element with position $\varphi N$ in the series, sorted in ascending order [6]. So, with the entire data set stored in this form, the quantile calculation operation will take O(1) time. At the same time, the operation of inserting a new value using the binary search algorithm will take O(log(n)) time and the following insertion will take up to O(n) time. This is still acceptable on small volumes but will take a lot of time on a large scale. This approach also has a large space complexity of O(n). Let us assume it is needed to calculate the request duration on a highly loaded application that processes 1000 requests per second. Then in 10 minutes, we will need to collect 600 000 values. It only applies to one metric, but there are typically dozens or hundreds of them. This will require a large amount of resources, which may be critical for highly loaded systems.

Alternative approaches involve the use of special data storage methods. T-Digest is a probabilistic data structure that allows obtaining the quantile values from a data stream without storing the entire set of values and their order [8]. It is generated by clustering values and storing the centroid – the average value, and its weight – the number of elements in a given cluster. Unlike classical clustering algorithms such as K-Means or Affinity Propagation, the clusters are limited in size in this data structure (configured by the end-user), so the data ranges included in different clusters may overlap [9 – 10].

New data is added as follows:

– a new value is added to the cluster with the closest centroid if its weight allows further accumulation, while the centroid and weight are recalculated to take into account the new value;

– if the found cluster cannot include the value, then a new cluster is created with that value as its centroid (larger weights are allowed closer to the middle, smaller ones closer to the edges);

– if the number of clusters exceeds a given configurable threshold, the least significant clusters are merged.

T-Digest uses interpolation between clusters to estimate quantile values. The accuracy, computational speed, and memory usage directly depend on the number of clusters. The complexity of adding new data is equal to O(log(n)).

The disadvantages include:

– the error can be significant on non-uniform distributions, especially in the middle quantiles;

– data arrival order may affect the quality of clustering;

– parameters configuring is required such as the maximum weight of clusters or clusters number threshold, as poor parameters selection can reduce accuracy or lead to high memory consumption.

The next approach is called Biased Quantiles [11]. It is a set of algorithms for estimating quantiles with a given accuracy and limited memory usage, which extends ideas given in the works [12 – 14]. Unlike T-Digest, this method does not allow obtaining the value of an arbitrary quantile. Instead, the desired quantiles and errors for each of them must be specified in advance before the start of data collection. Moreover, separate error values can be specified for different quantiles: an error of $\varepsilon = 0.05$ (a deviation of 5% from the true value) for the middle quantiles is quite acceptable, but is not suitable for the 0.99 quantile, while an error of $\varepsilon = 0.001$ would be suitable for any quantile, but would require storing more data. This compromise between memory usage and insertion time on the one hand and accuracy on the other hand is a crucial factor in this approach.

A binary tree is used to store compressed data. The space complexity is O(log(n)), an efficient event on large data sets. Each node of the tree stores the following data:

– value $v$ that represents the elements of a group;

– number of elements in the group (weight) $g$;

– an error or tolerance of the value $d$.

In the technical implementation, nodes are not stored in the form of a classical binary tree, but simply in the form of an ordered list by value *v* When a new value is added to the list, a new node is created with the corresponding value, and weight $g = 1$, $d$ is calculated depending on the difference between the weights of neighboring nodes. The data list is periodically compressed to control memory usage: if two neighboring nodes have a total error that does not exceed a given threshold, they are combined into a new one. Due to separate values of the error ε for different quantiles, more nodes will be stored in places where greater accuracy is required (usually at the edges). The amortized data insertion complexity is equal to O(log(log(n))).

This approach has the following disadvantages:

– achieving high accuracy requires more memory and computation time;

– the compression mechanism is more complicated than in T-Digest and is harder to implement.

A comparison of data insertion time (random real numbers in the interval [0; 100]) and memory usage for the considered algorithms is given in Table 1 and Table 2. Open-source implementations of algorithms [9] and [11] were used. Performance tests were conducted on an Apple MacBook with an Apple M1 Pro processor and 32 GB of RAM available. The insertion time was calculated as the average value of 100 test runs.

Table 1. Data insertion time comparison

| **Approach** | **Insertion time for 10 000 items** | **Insertion time for 100 000 items** | **Insertion time for 1 000 000 items** |
|---|---|---|---|
| Naive | 0.01 s | 0.89 s | 82.58 s |
| T-Digest | 0.23 s | 2.49 s | 26.82 s |
| Biased Quantiles | 0.05 s | 0.65 s | 10.59 s |

Table 2. Memory usage comparison

| **Approach** | **Memory usage for 10 000 items** | **Memory usage for 100 000 items** | **Memory usage for 1 000 000 items** |
|---|---|---|---|
| Naive | 700 kB | 4150 kB | 47000 kB |
| T-Digest | 350 kB | 450 kB | 540 kB |
| Biased Quantiles | 250 kB | 560 kB | 1230 kB |

Storing the entire data set (Naive approach) is only acceptable on very small volumes. On a large scale, T-Digest is more memory-efficient than Biased Quantiles but loses in time complexity for any amount of data. Server applications usually have relatively small processor resources allocated: dozens of different servers can be running on a single CPU core. So, the speed of data insertion will play a decisive role in order not wasting a large amount of processor time and not blocking the main service process. Therefore, it is more appropriate to use the Biased Quantiles approach to calculate the quantile values on the client side.

### 4.2. Real-time data handling

Any of the algorithms considered, including Biased Quantiles, cannot be directly applied when implementing the Summary client metric, since it uses all the accumulated data to calculate a certain quantile value. For instance, if data were collected for one or two hours, it would be impossible to say what the data distribution was during the last 10 minutes. A similar problem is observed in the third-party aioprometheus library mentioned above – the metric value becomes unrepresentative over time.

The data structure used in the Biased Quantiles algorithm does not contain information about the time of data insertion. Adding a timestamp directly to the compressed binary tree can significantly increase both the insertion time and the amount of memory used, and also complicate technical

implementation. It would be more appropriate to build some logic on top of this algorithm. Since the Biased Quantiles approach is fast for data insertion, the algorithm can be applied independently for several time ranges (called buckets), rotating them with a certain periodicity. To obtain quantile values, the "oldest" bucket containing the most historical data will be used, the remaining buckets will simply accumulate data and wait for the next rotations. The principle of operation of the described method is shown in Fig. 4 and Fig. 5.

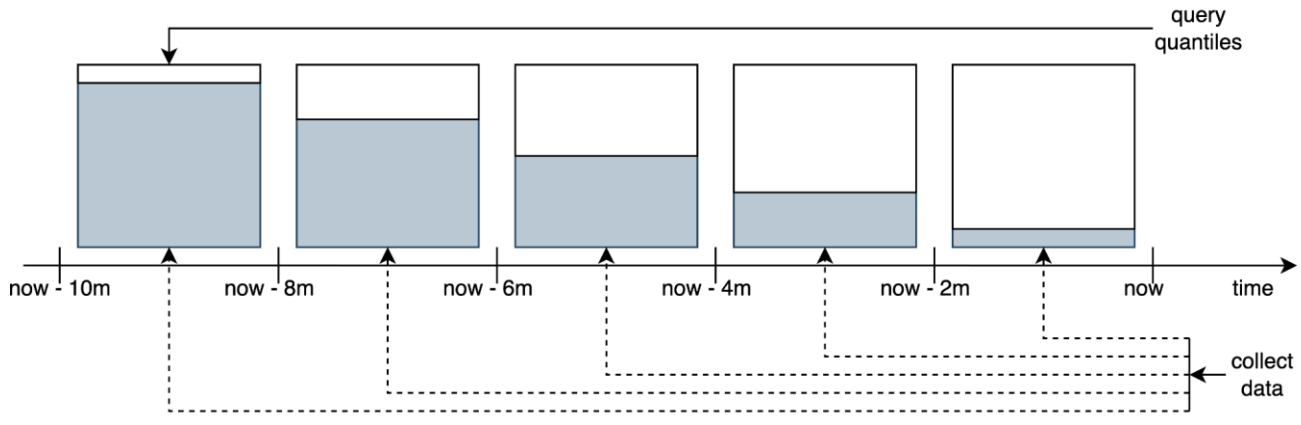


Fig. 4. The principle of collecting data over different time ranges

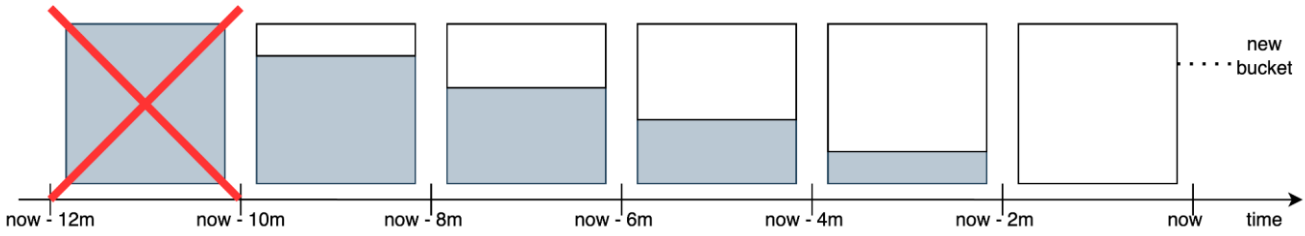


Fig. 5. The principle of periodic buckets rotation

The implemented method allows displaying collected data in dynamics, having the same order of time and space complexity as the original Biased Quantiles algorithm, thanks to the principle of periodic rotation.

### 4.3. Python extension implementation

The developed extension was named prometheus_summary. Class diagram is shown in Fig. 6. It includes the following custom classes (marked with gray color):

– "Invariant" – pair of quantile values and allowed error for it;

– "QuantileEstimator" – implements Biased Quantiles algorithm, described in detail in [11]. New data is collected by using the method "observe", quantile value can be obtained with the method query;

– "TimeWindowEstimator" – contains an array of "QuantileEstimator" instances, configured via "age_buckets" property. Time range is also variable via "max_age_seconds" parameter. Buckets rotation performed in the private method "rotate_buckets" (only if needed, based on the current timestamp). It is called under the hood in "query" and "observe" methods.

– "prometheus_summary.Summary" – extends the official library "prometheus_client.Summary" and overrides the "observe", "metric_init" and "child_samples" methods. "TimeWindowEstimator" instance is directly used here to collect data and obtain needed quantile values.

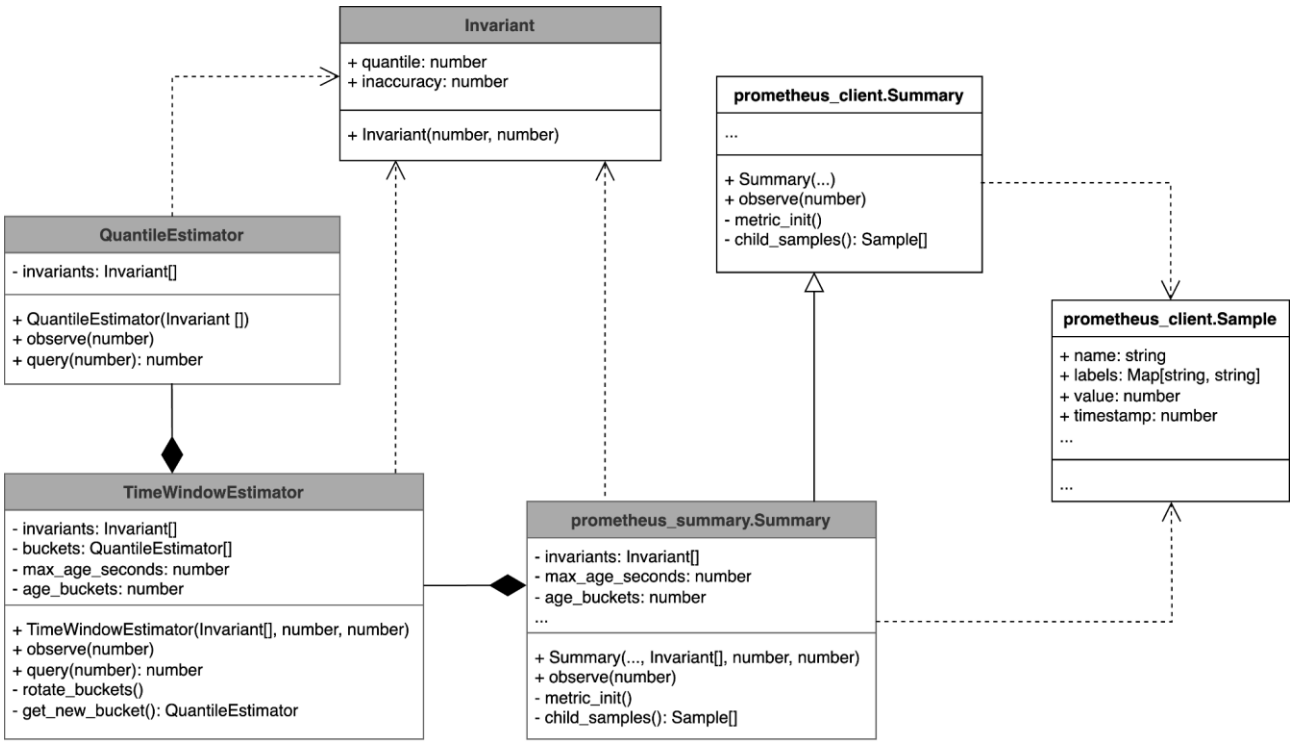


Fig. 6. Class diagram of the developed extension

## 5. Results of investigating collecting metrics across quantiles

The developed extension was published to GitHub [15] and PyPI (Python Package Index). So, it can be easily installed with the command “pip install prometheus-summary”. The interface is fully compatible with the official Prometheus client library, so no refactorings are need during integration for end-users.

Let us reproduce the experiment with numerical series over a certain time period from Section 2, with using the developed “prometheus-summary” library. Each point is the current minute value plus a real random value ranging from 0 to 10. The generated value is collected every 0.02 seconds. It means, 50 values are collected every second, which resembles a medium-loaded project. After the 3 hours of collecting, we can build the graph for quantile values at 10-minute intervals with PromQL query “sum(test_metric) by (quantile)”. The result is shown in Fig. 7. The graph is correct from a statistical point of view. Unlike in the aioprometheus library (shown in Fig. 3), the metric does not become constant after some period of time. Thanks to the principle of periodic buckets rotation, memory and CPU usage on the application process is low and stable. It is even non-visible compared to the full absence of metrics collecting.

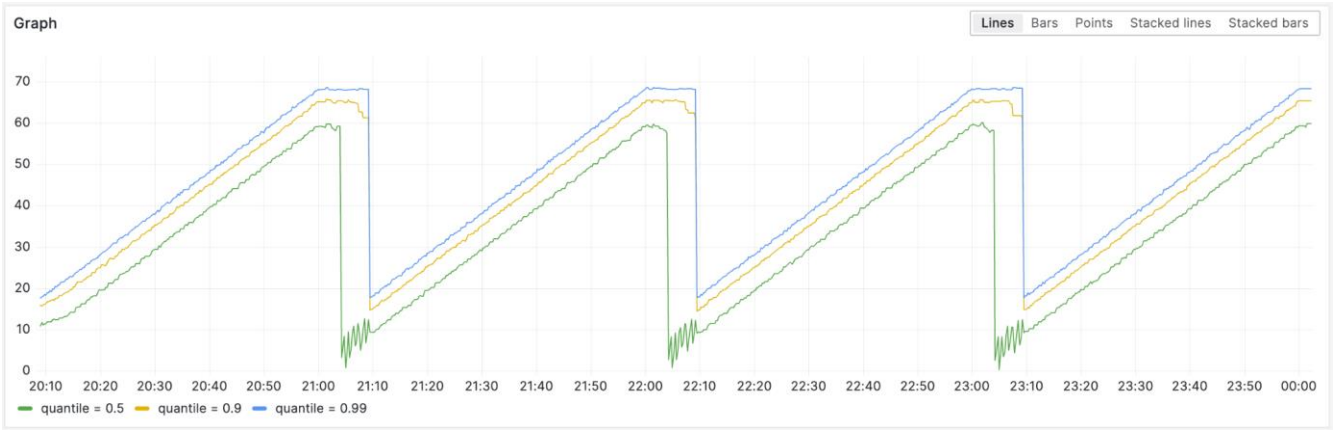


Fig. 7. The graph of series quantiles built with the developed Prometheus-summary

The novelty of the research is the adaptation of the existing quantile calculation algorithm for the official Prometheus library use case. Prometheus-summary is the first ever Python extension developed that performs exact client-side calculation of quantile values for the Summary metric. It can be easily used for both synchronous and asynchronous applications with an arbitrary load level.

## 6. Discussion of results collecting metrics across quantiles

The developed extension was integrated to a medium-loaded production project (about 10 requests per second). An example of the duration of two different requests across quantiles:

– slower – with a median ranging from 5 to 10 seconds – is shown in Fig. 8;

– faster – with a median of about 25 milliseconds – is shown in Fig. 9.

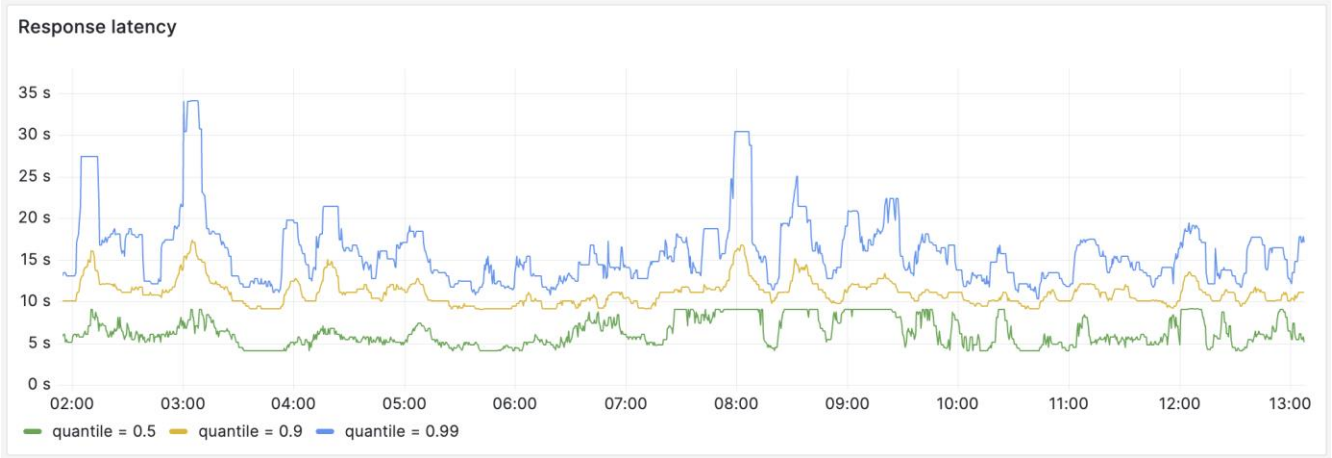


Fig. 8. The duration of the slower request across quantiles

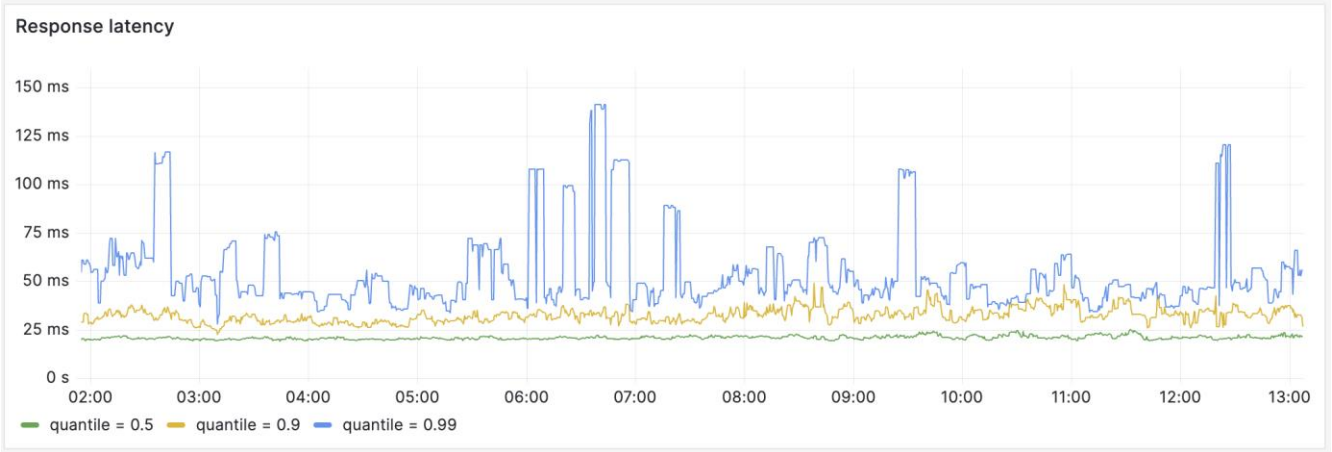


Fig. 9. The duration of the faster request across quantiles

As we can see, quantile values calculation on the client side is performed correctly on any data distribution with the single Summary metric instance in the code. Problems shown in Fig. 2 and Fig. 3 were not reproduced here.

It is possible to plot graphs with similar accuracy using the method proposed in the official documentation – via Histogram. However, it causes the following difficulties:

– a preliminary analysis of the data distribution is required to choose the correct segments;

– if the data is very variable, it will require a large number of segments (the library uses 12 segments by default, but to reproduce the above graphs it is needed about 24 segments), which will demand more server resources to store and handle it;

– separate types of metrics for different groups of requests can be created to save memory: fast, medium-duration, and slow, but then versatility is lost and the process of collecting indicators becomes more complicated.

Using the developed extension, the data collected will take up $qT$ of memory, where q is the number of quantiles (in this case $q = 3$), and $T$ is the number of times the metrics are scraped by the Prometheus server. In the case of Histogram, the collected indicator will take up $sT$ of memory, where s is the number of segments (for the above graphs, $s = 24$). It is easy to conclude that Summary will take up 8 times ($qT / sT = q / s = 24 / 3 = 8$) less memory than Histogram.

In the future, because of its universality, the developed library may be included directly in the popular Python web-frameworks like Django, Flask, FastAPI, etc. It may provide a wide range of useful metrics out-of-the box, without any additional code and preliminary analysis of possible value distribution. The library may also become the basis for further automated decision-making systems that are based on application metrics.

## Conclusion

As a result of the research, the analysis of algorithms for calculating quantile values on the client side was performed. The chosen algorithm was adapted for use with real-time data and first implemented as the Python extension for the official Prometheus library. It opens up opportunities for comprehensive monitoring of highly loaded systems in terms of both server resource usage and the quantity and quality of collected useful data. This solution can be easily implemented on large projects requiring continuous tracking of various metrics to ensure stable and uninterrupted service operation.

## References

[1] Prometheus. “Overview.” Accessed: Nov. 14, 2024. [Online]. Available: https://prometheus.io/docs/introduction/overview/.

[2] A. Mueen, E. Keogh, Q. Zhu, S. Cash, and B. Westover. “Exact Discovery of Time Series Motifs,” in *Proc. of the SIAM International Conference on Data Mining (SDM)*, Sparks, NV, USA, pp. 473–484, 2009, https://doi.org/10.1137/1.9781611972795.41.

[3] C. Wang et al., “Apache IoTDB: time-series database for internet of things,” in *Proc. VLDB Endow*, vol. 13, no. 12, pp. 2901–2904, 2020, https://doi.org/10.14778/3415478.3415504.

[4]S. Alhusain. “Predicting Relative Thresholds for Object Oriented Metrics,” Cornell University, p. 9, 2021, https://doi.org/10.48550/arXiv.2103.11442.

[5] Prometheus. “Metric types.” Accessed: Nov. 14, 2024. [Online]. Available: https://prometheus.io/docs/concepts/metric_types/.

[6] Z. Chen and A. Zhang, “A Survey of Approximate Quantile Computation on Large-Scale Data,” in *IEEE Access*, vol. 8, pp. 34585–34597, 2020, https://doi.org/10.1109/ACCESS.2020.2974919.

[7] L. Chen and A. Dobra. “Histograms as statistical estimators for aggregate queries”, in *Information Systems*, vol. 38, no. 2, pp. 213–230, 2013, https://doi.org/10.1016/j.is.2012.08.003.

[8] F. Chen, D. Lambert, and J. Pinheiro. “Incremental Quantile Estimation for Massive Tracking, ” in *Proc. of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Boston, MA, USA, pp. 516–522, 2000, https://doi.org/10.1145/347090.347195.

[9] T. Dunning and O. Ertl. “Computing Extremely Accurate Quantiles Using t-Digests,” Cornell University, p. 22, 2019, https://doi.org/10.48550/arXiv.1902.04023.

[10]T. Dunning. “The t-digest: Efficient estimates of distributions,” in *Software Impacts*, vol. 7, p. 100049, 2021, https://doi.org/10.1016/j.simpa.2020.100049.

[11] G. Cormode, S. Muthukrishnan, F. Korn, and D. Srivastava. “Effective Computation of Biased Quantiles over Data Streams,” in *Proc. of the 21st International Conference on Data Engineering (ICDE’05)*, Tokyo, Japan, pp. 20–31, 2005, https://doi.org/10.1109/ICDE.2005.55.

[12] M. Greenwald and S. Khanna. “Space-Efficient Online Computation of Quantile Summaries,” in *Proc. of the ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA, USA, pp. 58–66, 2001, https://doi.org/10.1145/375663.375670.

[13] M. Ajtai, T. S. Jayram, R. Kumar, and D. Sivakumar. “Approximate counting of inversions in a data stream,” in *Proc. of the 34th Annual ACM Symposium on Theory of Computing (STOC’02)*, New York, NY, USA, pp. 370–379, 2002, https://doi.org/10.1145/509907.509964.

[14] X. Lin, H. Lu, J. Xu, and J. X. Yu. “Continuously maintaining quantile summaries of the most recent N elements over a data stream,” in *Proc. of the 20th International Conference on Data Engineering (ICDE’04)*, Boston, MA, USA, pp. 362–373, 2004, https://doi.org/10.1109/ICDE.2004.1320011.

[15] GitHub. “Prometheus-summary.” Accessed: Nov. 23, 2024. [Online]. Available: https://github.com/RefaceAI/prometheus-summary.

УДК 004.42

# ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ЗБОРУ ТА АНАЛІЗУ МЕТРИК У ВИСОКОНАВАНТАЖЕНИХ ЗАСТОСУНКАХ НА БАЗІ СИСТЕМИ МОНІТОРИНГУ PROMETHEUS

**Інна Стеценко**
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”, Київ, Україна
https://orcid.org/0000-0002-4601-0058

**Антон Миронюк**
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”, Київ, Україна
https://orcid.org/0009-0003-7212-2816

В даній роботі підкреслюється важливість збору метрик в процесі роботи застосунків для раннього виявлення потенційних проблем. Одноосібним лідером у цій сфері є система моніторингу *Prometheus*, що у поєднанні із *Grafana* – платформою для візуалізації зібраних даних у формі численних графіків, стає незамінним інструментом в арсеналі програмістів та *SRE*-спеціалістів. Втім, просто середнє значення певного показника досить часто буває нерепрезентативним, адже воно не відображає комплексної картини. Натомість корисно збирати метрики в розрізі різних квантилів на тривалому проміжку часу для виявлення навіть одиничних нестабільностей, проте використання стандартних засобів в *Python*-екосистемі може вимагати надто великої кількості серверних ресурсів та тривалого попереднього аналізу, що з фінансової сторони може бути досить затратним для бізнесів. Саме тому актуальною є розробка нового підходу для збору та аналізу метрик у високонавантажених сервісах на базі *Prometheus*.

Мета дослідження полягає у створенні додаткових можливостей для збору та аналізу метрик у високонавантажених застосунках на основі системи моніторингу *Prometheus* шляхом обрахування квантилів на клієнтській стороні.

Було проведено теоретичний огляд наявних підходів для обрахування значення квантилів на великих обсягах даних. Також представлено їх порівняльну характеристику в контексті швидкості роботи та обсягу використовуваної пам’яті, виміряну експериментальним шляхом на різних обсягах даних. Обраний метод було адаптовано для використання із потоком даних у реальному часі та реалізовано у вигляді *Python*-розширення для офіційної бібліотеки *Prometheus*. Це відкриває можливості для комплексного моніторингу високонавантажених систем з погляду як використання серверних ресурсів, так і кількості та якості зібраних корисних даних. Дане рішення може бути з легкістю втілене на великих проєктах, які вимагають постійного відстеження багатьох метрик задля забезпечення стабільної та безперебійної роботи.

**Ключові слова:** метрики, квантиль, високонавантажені застосунки, моніторингова система *Prometheus, Python*.