Information systems. Computer sciences. Issues of information security
Информационные системы. Информатика. Проблемы информационной безопасности

UDC 004.657
https://doi.org/10.32362/2500-316X-2024-12-3-7-24
EDN BNQNDI

RESEARCH ARTICLE

Evaluation of connection pool *PgBouncer* efficiency for optimizing relational database computing resources

Anton S. Boronnikov @,
Pavel S. Tsyngalev,
Victor G. Ilyin,
Tatiana A. Demenkova

MIREA – Russian Technological University, Moscow, 119454 Russia
@ Corresponding author, e-mail: boronnikov-anton@mail.ru

Abstract

Objectives. The aim of the research is to investigate the possibilities of using the *PgBouncer* connection pool with various configurations in modern database installations by conducting load testing with diverse real-world like scenarios, identifying critical metrics, obtaining testing results, and interpreting them in the form of graphs.

Methods. The research utilized methods of experimentation, induction, testing, and statistical analysis.

Results. The main features, architecture and modes of operation of the *PgBouncer* service are considered. Load testing was carried out on a virtual machine deployed on the basis of an open cloud platform with different configurations of computing resources (CPU, RAM) and according to several scenarios with different configurations and different numbers of balancer connections to the database, during which the following main indicators were investigated: distribution of processor usage, utilization of RAM, disk space, and CPU. The interpretation of the data obtained and the analysis of the results obtained by highlighting critical parameters are performed. On the basis of results analysis, conclusions and recommendations are formulated on the use of a connection balancer in real high-load installations for optimizing the resources utilized by the server on which the database management system (DBMS) is located. A conclusion is presented on the usefulness of using the *PgBouncer* query balancer along with proposed configuration options for subsequent use in real installations.

Conclusions. The degree of influence of the use of the *PgBouncer* connection balancer on the performance of the system as a whole deployed in a virtualized environment is investigated. The results of the work showed that the use of *PgBouncer* allows significantly optimization of the computing resources of a computing node for a DBMS server, namely, load on the CPU decreased by 15%, RAM—by 25–50%, disk subsystem—by 20%, depending on the test scenarios, the number of connections to the database, and the configuration of the connection balancer.

Keywords: PgBouncer, PostgreSQL, connection pool, balancer, databases, optimization, monitoring, virtual machines, cloud technologies

• **Submitted:** 13.06.2023 • **Revised:** 06.12.2023 • **Accepted:** 09.04.2024

For citation: Boronnikov A.S., Tsyngalev P.S., Ilyin V.G., Demenkova T.A. Evaluation of connection pool *PgBouncer* efficiency for optimizing relational database computing resources. *Russ. Technol. J.* 2024;12(3):7–24. https://doi.org/10.32362/2500-316X-2024-12-3-7-24

Financial disclosure: The authors have no a financial or property interest in any material or method mentioned.

The authors declare no conflicts of interest.

НАУЧНАЯ СТАТЬЯ

Оценка эффективности балансировщика соединений *PgBouncer* для оптимизации вычислительных ресурсов реляционных баз данных

А.С. Боронников [@], **П.С. Цынгалёв**, **В.Г. Ильин**, **Т.А. Деменкова**

МИРЭА – Российский технологический университет, Москва, 119454 Россия
[@] *Автор для переписки, e-mail: boronnikov-anton@mail.ru*

Резюме

Цели. Целью работы является исследование возможностей использования балансировщика подключений *PgBouncer* с различными конфигурациями в современных инсталляциях баз данных (БД) путем проведения нагрузочного тестирования с различными сценариями, максимально приближенными к реальной нагрузке, определение критичных показателей, получение результатов тестирования и интерпретация их в виде графиков.

Методы. В ходе исследования использовались методы эксперимента, индукции, тестирования и статистического анализа.

Результаты. Рассмотрены основные возможности, архитектура и режимы работы сервиса *PgBouncer*. Проведено нагрузочное тестирование на виртуальной машине, развернутой на базе открытой облачной платформы, с различной конфигурацией затрачиваемых вычислительных ресурсов – центрального процессора (CPU) и оперативной памяти (RAM) и использованием нескольких сценариев с разной конфигурацией и разным количеством подключений балансировщика к БД. В ходе тестирования были исследованы основные показатели: распределение использования процессора, утилизация оперативной памяти, дискового пространства и центрального процессора. Выполнены интерпретация полученных данных и анализ полученных результатов путем выделения критических параметров. Сформулированы выводы и рекомендации по использованию балансировщика подключения в реальных высоконагруженных инсталляциях для оптимизации утилизируемых ресурсов сервером, на котором расположена система управления базами данных (СУБД). Сформировано заключение о полезности использования балансировщика запросов *PgBouncer* и предложены варианты конфигурации для последующего использования в реальных инсталляциях.

Выводы. Исследована степень влияния использования балансировщика соединений *PgBouncer* на производительность системы в целом, развернутой в виртуализированной среде. Результаты работы показали, что применение *PgBouncer* позволяет существенно оптимизировать затрачиваемые вычислительные ресурсы вычислительного узла под сервер СУБД, а именно: уменьшилась нагрузка на CPU на 15%, на RAM – на 25–50%, на дисковую подсистему – на 20%, в зависимости от сценариев тестов, количества подключений к БД, конфигурации балансировщика подключений.

Ключевые слова: PgBouncer, PostgreSQL, пуллер, балансировщик, база данных, оптимизация, мониторинг, виртуальная машина, облачные технологии

• **Поступила:** 13.06.2023 • **Доработана:** 06.12.2023 • **Принята к опубликованию:** 09.04.2024

Для цитирования: Боронников А.С., Цынгалёв П.С., Ильин В.Г., Деменкова Т.А. Оценка эффективности балансировщика соединений *PgBouncer* для оптимизации вычислительных ресурсов реляционных баз данных. *Russ. Technol. J*. 2024;12(3):7–24. https://doi.org/10.32362/2500-316X-2024-12-3-7-24

Прозрачность финансовой деятельности: Авторы не имеют финансовой заинтересованности в представленных материалах или методах.

Авторы заявляют об отсутствии конфликта интересов.

INTRODUCTION

Databases (DB) are an integral part of modern applications. They are used to store, manage, and process large scopes of information. However, one of the main problems faced by applications is managing multiple DB connections.

Connection to a DB involves the process of establishing a link between a client application and the DB server. Since each client application establishes its own connection to the DB, excessive loads on the DB server can decrease the application performance. In addition, each DB connection requires certain resources such as memory and CPU time. If a significant number of client applications establish a DB connection at the same time, it may overload the server and degrade the application performance.

In order to solve this problem, the DB management system (DBMS) can be optimized by configuring its parameters at the infrastructure startup stage [1–3] or using third-party services such as DB connection balancers. Such tools manage client connections in such a way as to maximize the use of DB server resources, thus improving application performance. There are several types of balancers [4]. This article discusses the main features of a tool that belongs to the application-level balancing type known as a *connection pooler* or *pooler*.

1. DB QUERY PATH

Implementation of the pooler leads to significant changes in working with the DB. In order to notice them, it is necessary to study the standard query passing route. In the usual client-server architecture, the following standard interaction principle takes place as shown in Fig. 1.

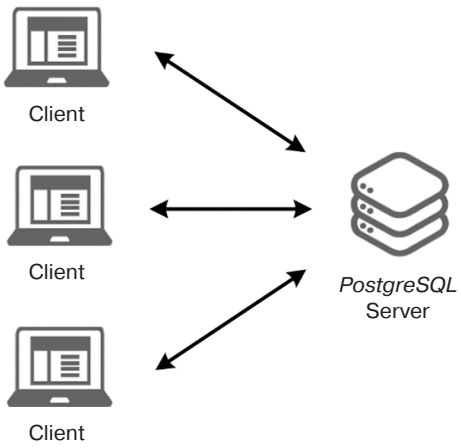

Fig. 1. Regular client-server connection[1]

[1] *PostgreSQL*. https://www.postgresql.org/. Accessed April 18, 2023.

When a new session is established, the client application requests a connection to the server and goes through the authentication process. The server responds by creating a separate system process to handle the connection and session operation. The initialization of the session state is based on various configuration parameters defined at server, DB, and user levels. Within a single session, the client performs the required operations. The operation continues until the client terminates the session by disconnecting. After the session is terminated, the server destroys the corresponding system process responsible for processing this session.

The main disadvantages of a regular client-server connection can be highlighted as follows:

1) creating, managing and deleting connection processes takes time and consumes resources;
2) as the number of connections on the server increases, so does the need for resources to manage them. In addition, memory utilization on the server increases as clients perform operations;
3) since a single session serves only one client, clients can change the state of a DB session and expect those changes to persist in subsequent transactions.

When using a pooler, clients connect to the puller that has already established a connection to the server (Fig. 2). This changes the model of the standard connection principle to a client-proxy-server architecture.

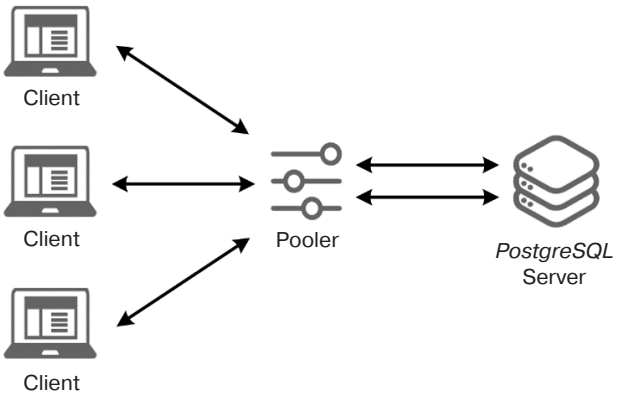

Fig. 2. Client-proxy-server connection

Now the connection of a client to a server is independent of the lifetime of the connection and the process on the server. The pooler is responsible for accepting and managing connections from the client, establishing and maintaining connections to the server, and assigning server connections to client connections.

PostgreSQL [5–9] is an excellent choice in terms of the quality of the DBMS. One of the main factors is *PostgreSQL*'s status as open-source software. There are several decent *PostgreSQL* connection

poolers [6, 10–12] such as *PgBouncer*[2], *Pgpool-II*[3], and *Odyssey*[4]. In this paper, the main features, architecture, and modes of operation of *PgBouncer* are considered.

2. *PgBouncer* CONNECTION POOL

PgBouncer is a pooler that allows you to manage connections to a *PostgreSQL* DB. It works as a proxy server that processes DB connection requests and redirects them to the appropriate server. *PgBouncer* can be installed on the same machine as *PostgreSQL* or on a separate one.

This pooler widely used in many *PostgreSQL*-based applications is for solving various tasks related to performance, scalability and security. This balancer is actively used in products of such large companies as Alibaba[5], Huawei[6], Instagram[7] (banned in the Russian Federation), Skype[8], as well Russian companies[9] such as Yandex[10], Avito[11], Sberbank[12], Gazpromneft[13], etc.

One of the main tasks of *PgBouncer* is connection management. This allows the creation of connection pools that can be used by multiple clients, which reduces load on the DB server and improves application performance.

2.1. Architecture

In the official documentation of *PgBouncer* there is no description of the balancer architecture. After analyzing the libraries of this pooler and investigating its functionality, we propose a *PgBouncer* balancer architecture based on our own reverse-engineering (Fig. 3).

Listener plays an important role in handling client connections to *PostgreSQL* DB. Providing an entry point for client connections (called a socket), it acts as an intermediary between the client and the server. The listener still includes a protocol that defines the format of data exchange between the client and the server over the socket. *PgBouncer* uses the same protocol as *PostgreSQL*, but has its own additional extensions and commands.

Authentication provides verification of the client's authenticity when an attempt is made to connect to it. Various methods such as md5, trust, plain, cert, etc. are supported.

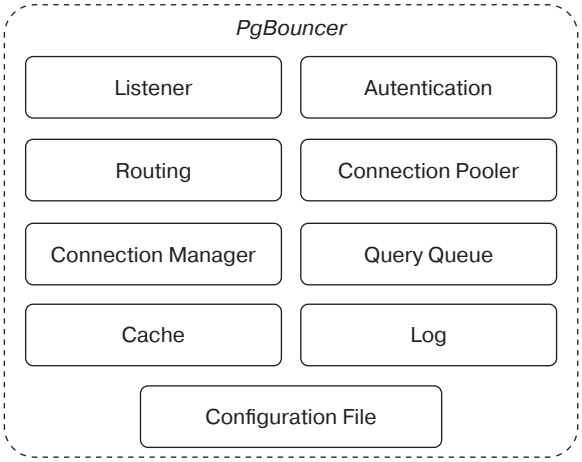

Fig. 3. *PgBouncer* architecture

Routing determines the modes of pooler operation. There are four types: session pooling, transaction pooling, statement pooling, and combined pooling. They are discussed in more detail in Section 2.2.

Connection pool is a set of available connections that can be used by the client to perform operations in the DB. If a connection is free, the client can take it from the pool. If there is no connection, the pooler can create a new one.

Connection manager is responsible for managing the lifecycle of each connection in the pool. It tracks the state of each active connection to the DB, including opening, closing and reusing free connections.

Query queue manages queries coming from clients when all connections are busy. This ensures fair processing of queries and avoids DB blocking or overloading. The query queue has customizable parameters that enable controlling the number and waiting time of queries in the queue.

Cache is an additional component that stores the results of previous queries so that when a query is executed again, the results are returned directly from the cache instead of accessing the DB. Cache saves time and resources for query execution, especially if the queries are often repeated and their results do not change.

Event log keeps a record of events such as establishing and breaking connections, executing requests and other operations, etc. It can be used to analyze and track the operation of the pooler, detect various errors and

[2] Official documentation *PgBouncer*. https://www.pgbouncer.org/. Accessed April 02, 2023.

[3] *Pgpool* Wiki. https://pgpool.net/mediawiki/index.php/Main_Page. Accessed April 15, 2023.

[4] *Odyssey* – Yandex Technologies. https://yandex.ru/dev/odyssey/ (in Russ.). Accessed April 15, 2023.

[5] https://www.alibaba.com/. Accessed April 15, 2023.

[6] https://www.huawei.com/. Accessed April 15, 2023.

[7] https://www.instagram.com/. Accessed April 15, 2023.

[8] https://www.skype.com/ru/ (in Russ.). Accessed April 15, 2023.

[9] Why the largest companies in Russia and the world choose Postgres. Results of PgConf.Russia 2017. http://www.interface.ru/home.asp?artId=39028 (in Russ.). Accessed April 10, 2023.

[10] https://yandex.ru/ (in Russ.). Accessed April 15, 2023.

[11] https://www.avito.ru/ (in Russ.). Accessed April 15, 2023.

[12] http://www.sberbank.ru/ (in Russ.). Accessed April 15, 2023.

[13] https://www.gazprom-neft.ru/ (in Russ.). Accessed April 15, 2023.

Table 1. Comparison of *PgBouncer* operating modes

Comparison of *PgBouncer* operating modes	Isolation of transactions	Connection pool	Capacity
Session mode	Full	Assigned for the duration of the session	Decrease due to creation and deletion of the connections
Transaction mode	Full	Assigned for the duration of the transaction	Decrease due to creation and deletion of the connections
Statement mode	Partial	Assigned for the duration of the query	Increase due to reuse of the connections
Combined mode	Balance	Depending on the type of the query	Balance between capacity and insulation

warnings, monitor performance, and perform debugging activities.

This is followed directly by the *configuration file*, where settings can be configured and specified for all the architecture components discussed in this section.

When these elements are properly configured, they ensure efficient use of server resources, improve performance and performance of applications that use *PostgreSQL*.

2.2. Operating modes

Operating modes determine how the pooler will manage connections. In different modes, *PgBouncer* can have different effects on system performance and functionality.

Session mode, representing the standard approach, consists of assigning a single server connection to each client for as long as the client remains connected. When a client disconnects, this server connection is returned back to the pool. This is the default method of operation. The mode can be useful for applications that have many client requests that are not frequent, but are executed in long sessions.

Transaction mode means that the client is assigned a connection to the server only for the duration of the transaction. When transaction completion is detected, *PgBouncer* returns the connection back to the pool. This mode can be useful in applications that have many short transactions.

Statement mode is the most aggressive approach, which assumes that the connection to the server will be returned to the pool immediately after each request is completed. In this mode, transactions with multiple statements are prohibited. This mode can be useful for applications that have many repetitive queries or use queries with the same structure.

Combined mode is an approach that combines transaction and operator modes. *PgBouncer* will use statement mode for queries that do not start a new transaction, and transaction mode for queries that do start a new transaction. This method can be useful for applications that run many repetitive queries and require transactions. It can also be effective for applications that have a large number of unique queries, but where transactions may be repeated.

PgBouncer operation modes were compared according to the following criteria: transaction isolation, connection pooling and performance. The results of the study are shown in Table 1.

2.3. Features of use

Some key features of *PgBouncer* will be emphasized in order to avoid mistakes when working with it.

The first key feature is the limitation on query types. Some queries, such as DB creation or deletion, cannot be routed through the pool and must be executed directly on the *PostgreSQL* server. It is also worth considering the configuration of the DBMS itself, since some parameters, such as those related to caching, may affect the performance of the pooler itself. In this case, additional configuration of the *PostgreSQL* server will be required for optimal operation in conjunction with the pooler.

In order to avoid pooler overloading, it is necessary to control the size of the balancer connection pool otherwise overflow may occur to exhaust system resources such as central processing unit (CPU) and random-access memory (RAM) utilization. An overflowed pool can cause performance degradation or application failures.

When configuring extensions, it should be taken into account that some of them may not be compatible with *PgBouncer*, as they may create their own connections to the *PostgreSQL* server, which will harm the performance of the system as a whole. When using this balancer, it is also necessary to ensure that it is compatible with other

tools and technologies used in the application, such as the ORM-framework[14] used.

Version support will be required when using with *PgBouncer*, since some versions of the pooler may not support the latest versions of *PostgreSQL*.

3. TESTING AND EVALUATION OF THE RESULTS OBTAINED

Testing was conducted on two virtual machines (VM) with the following characteristics:

- VM with DBMS: 8 virtual central processing units (vCPU), 16 GB of random access memory (RAM), external SSD 32 GB (under system), 500 Mbps, 320 IOPS[15], 120 GB (under DB), 500 Mbps, 1200 IOPS, Ubuntu 22.04 operating system (OS);
- VM with the pooler: 2 vCPU, 4 GB RAM, 120 GB SSD external drive, 500 Mbps, 1200 IOPS, Ubuntu 22.04 OS.

In order to collect and display metrics, the software *pgwatch2*[16], *Grafana*[17] and *PostgreSQL* were used. These were run in a docker container [13–15]. 2 GB of RAM, 1 vCPU was allocated for system performance. During testing, the performance of the test bench VMs was increased, since some tests utilized all available resources.

Testing was performed using several connection scenarios: directly to the DB and through the pooler. The pooler was set to session mode. The scenarios themselves included a gradual increase in the number of connections (100, 500, 1000) and query complexity with an active session size of 10 min.

The queries were of the following nature:

- simple queries to an empty DB;
- crud queries (create, read, update, delete) with application of temporary tables to the DB containing test data.

The following metrics were highlighted for the analysis:

1. CPU utilization distribution:
   - idle—free resources;
   - user—expenditures on the use of the system by users;
   - system—system expenditures;
   - iowait—waiting on the disk subsystem;
   - other—other CPU operations;
   - irqs—CPU core interruptions.
2. Utilization of CPU.
3. Utilization of RAM.
4. Utilization of the disk subsystem (disk).

3.1. Read queries

As a part of this testing, a read request to the DB (obtaining the DBMS version) was performed in parallel.

This test was chosen to evaluate the session's impact on DB resources as fairly as possible (it is most strongly reflected in RAM utilization).

Metrics for the direct connections with simple queries are depicted in Fig. 4 (100 connections), Fig. 5 (500 connections), Fig. 6 (1000 connections), where the time segments shown in the graph reflect the change of parameters during the testing period in real time in the *hours:minutes* format.

Metrics for connections via pooler with simple queries are shown in Fig. 7 (100 connections), Fig. 8 (500 connections), Fig. 9 (1000 connections).

During testing, an anticipated strong impact of idle connections on the resources reserved for the DB was noted.

If CPU and disk utilization can be attributed to the error and influence of external factors, RAM should be considered in more detail. The following utilization was obtained for the tests:

1) 100 connections, 230 MB (2.3 MB/connection);
2) 500 connections, 1180 MB (2.36 MB/connection);
3) 1000 connections, 1810 MB (1.81 MB/connection).

RAM utilization rates when tested via pooler were extremely low and did not fluctuate depending on the number of connections, remaining at an extremely low level (~30 MB on all tests).

It is also worth noting that a single connection had the greatest impact on the resources consumed when there were 500 parallel connections.

[14] Object relation mapping is a programming technology that connects DBs with the concepts of object-oriented programming languages, creating a virtual object DB.

[15] IOPS—input/output operations per second.

[16] PGWatch: Optimized PostgreSQL monitoring. https://pgwatch.com. Accessed April 15, 2023.

[17] Grafana Labs. https://grafana.com. Accessed April 12, 2023.

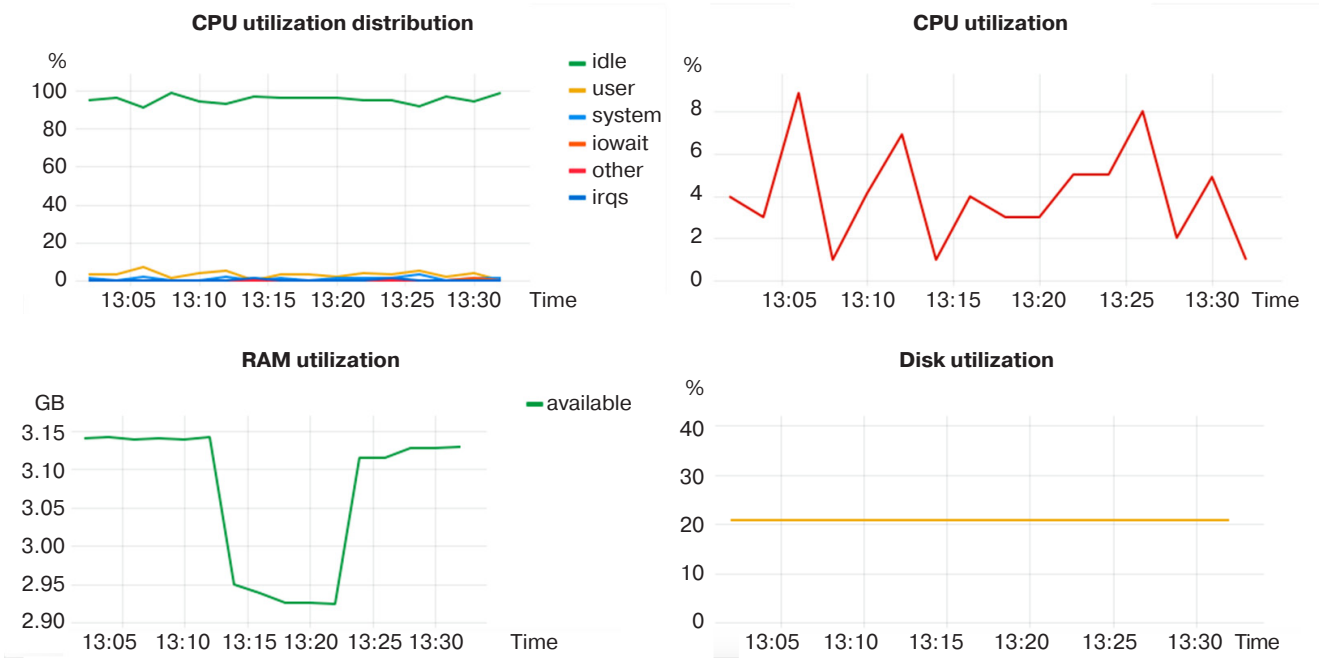

Fig. 4. Performing load testing with simple queries with 100 connections directly to an empty DB

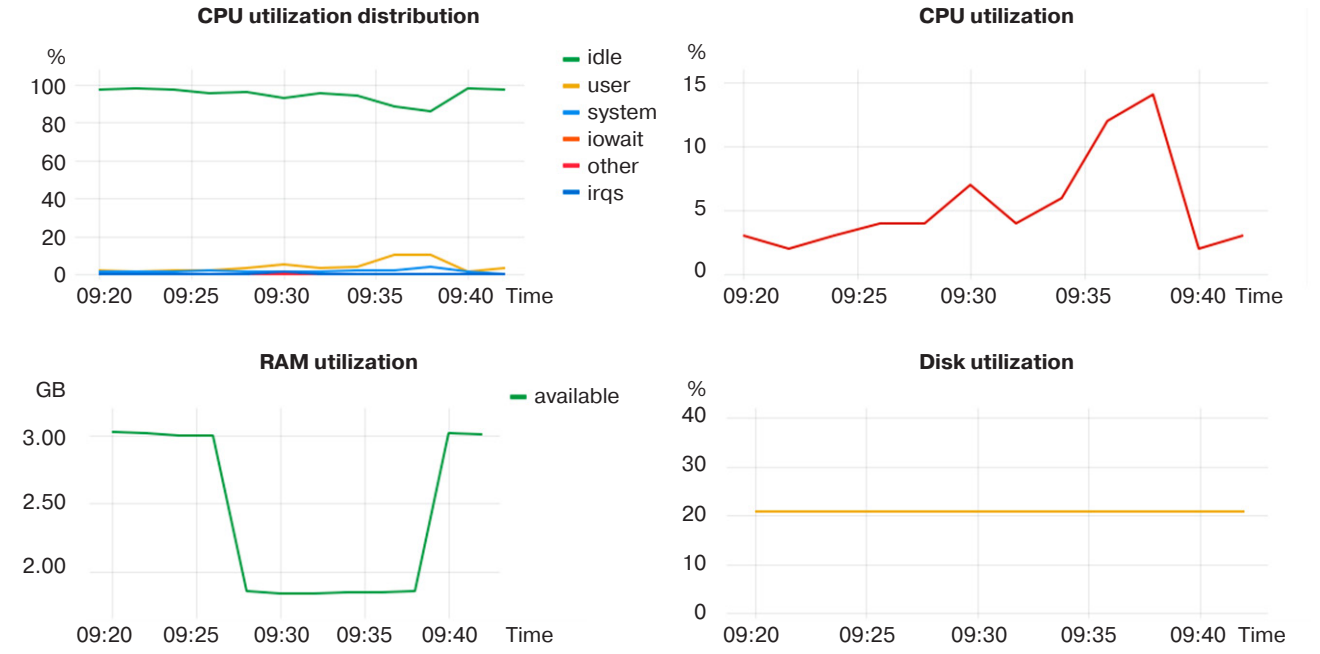

Fig. 5. Load testing with simple queries with 500 connections directly to an empty DB

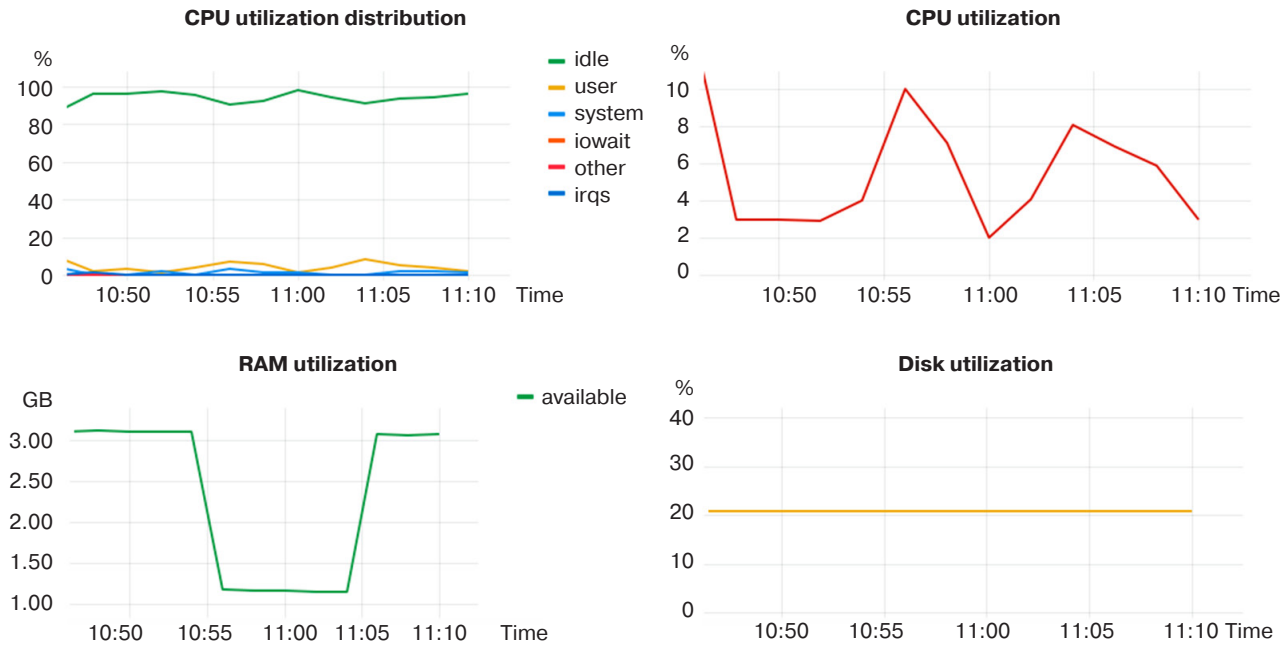

Fig. 6. Performing load testing with simple queries and 1000 connections directly to an empty DB

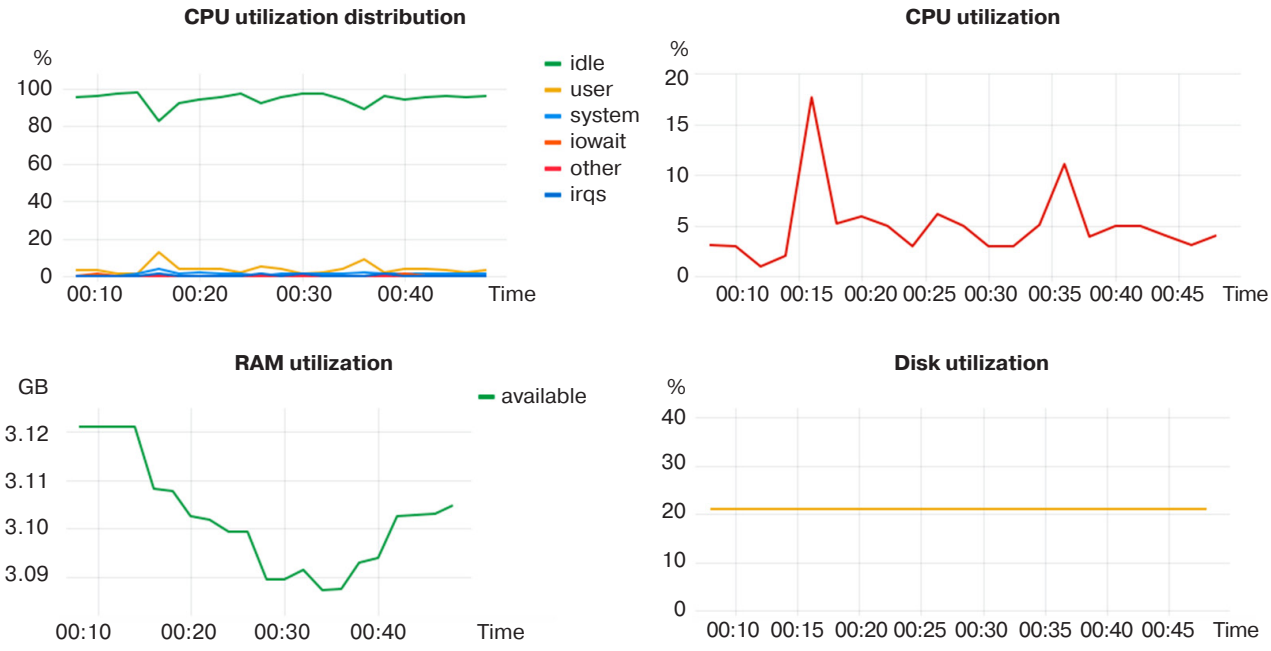

Fig. 7. Load testing with simple queries with the number of 100 connections through the pooler to an empty DB

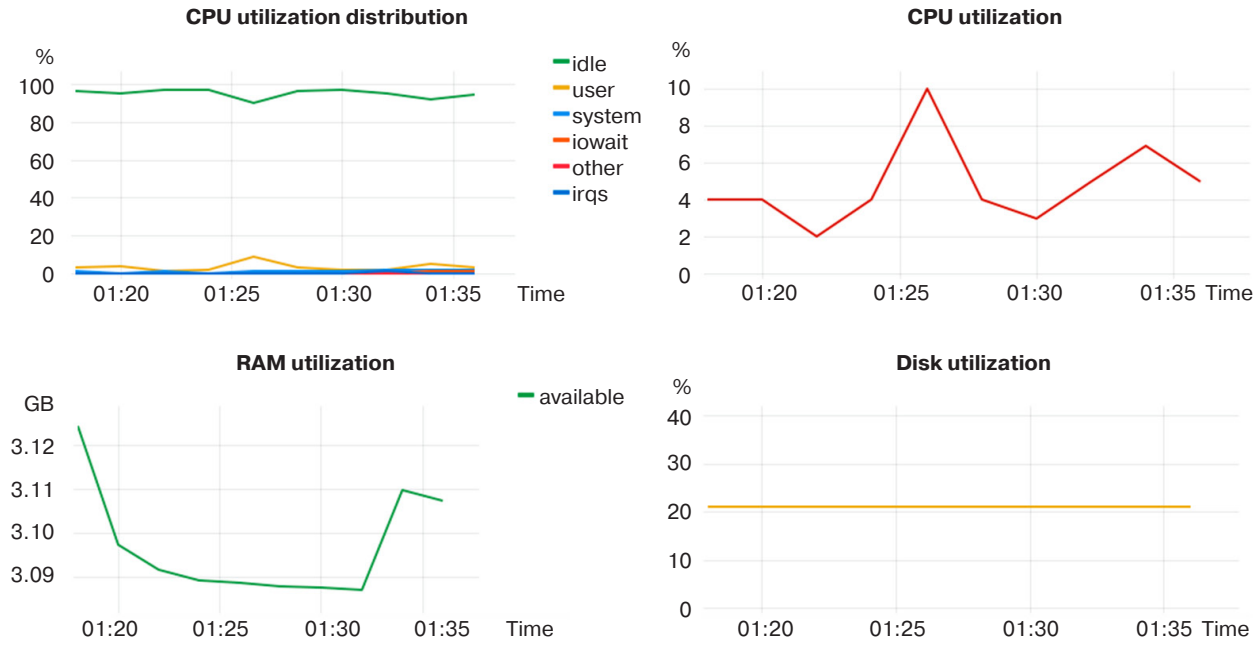

Fig. 8. Performing load testing with simple queries with the number of 500 connections through the pooler to an empty DB

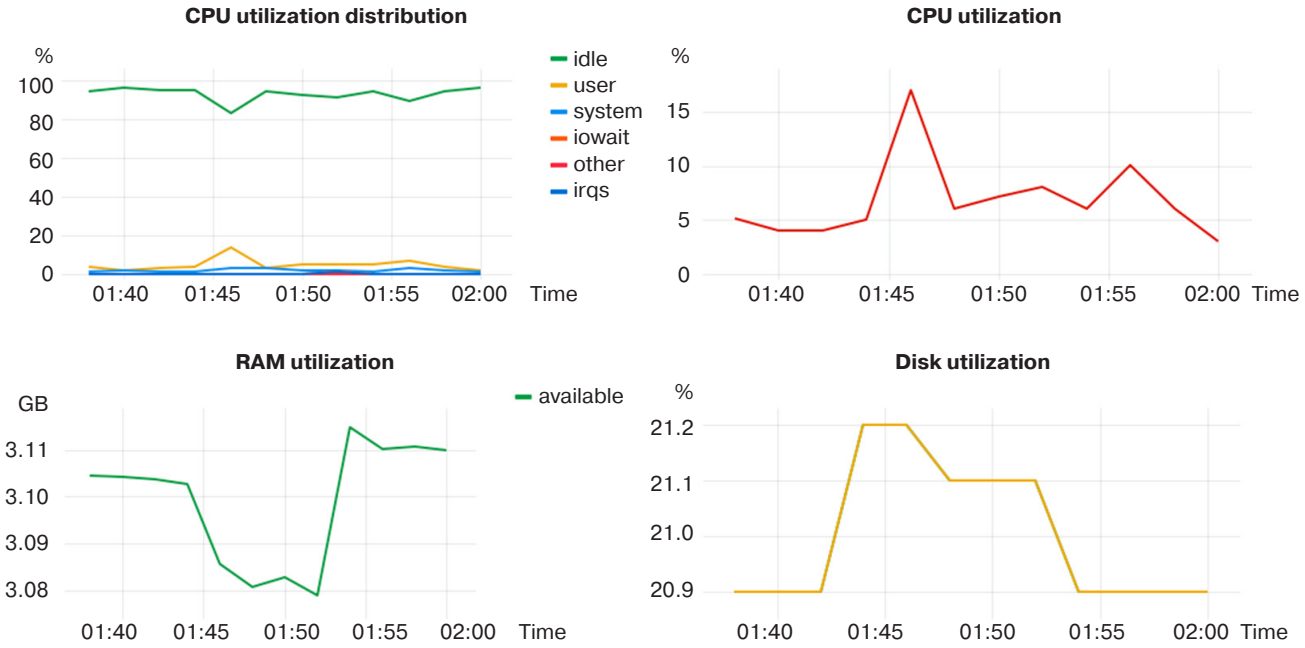

Fig. 9. Carrying out load testing with simple queries with 1000 connections through the pooler to an empty DB

3.2. Complicated queries

Within the framework of this testing, a query for writing to the DB, creating, filling, deleting a table (2 columns, 1000000 rows, with the *text* value type) was executed in parallel.

This test was chosen because it allows us to evaluate possible CPU and disk resource savings when using the request pooler.

The metrics for direct connections by complex queries are depicted in Fig. 10 (100 connections), Fig. 11 (500 connections), and Fig. 12 (1000 connections), where the time segments shown in the graph represent the change in metrics during the testing period in real time in the *hours:minutes* format.

Metrics for connections via pooler with complex queries are shown in Fig. 13 (100 connections), Fig. 14 (500 connections), and Fig. 15 (1000 connections).

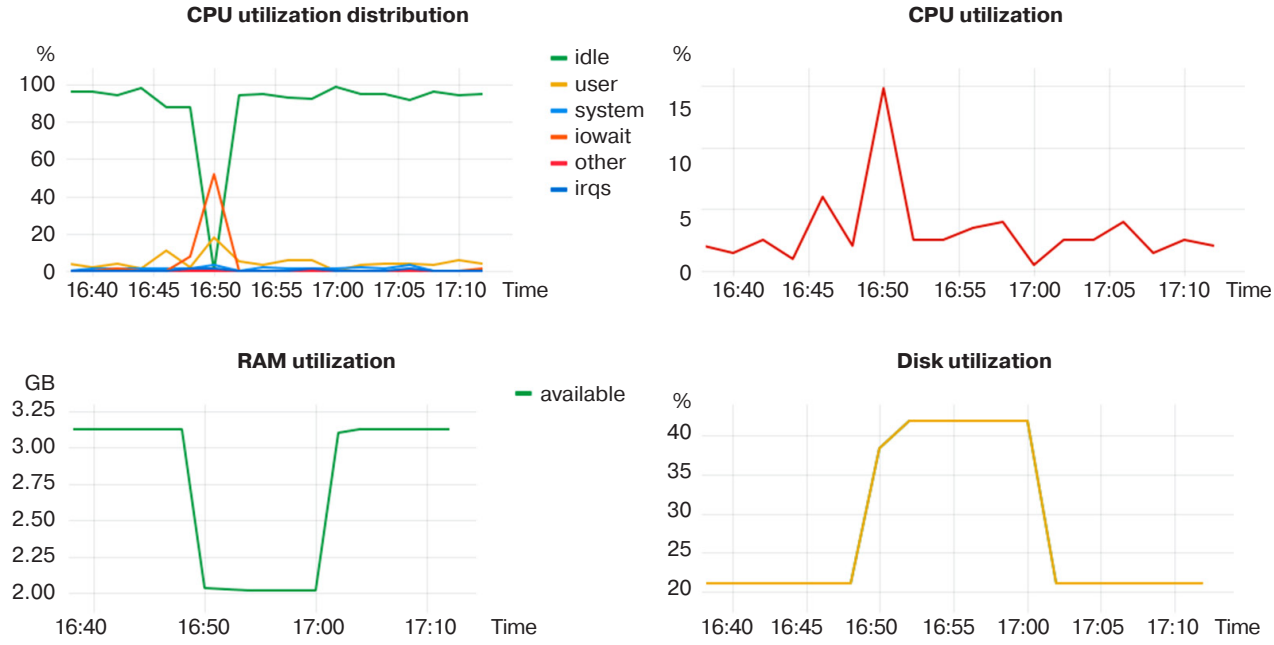

Fig. 10. Load testing with complicated queries using temporary tables and the number of 100 connections to the DB filled with test data

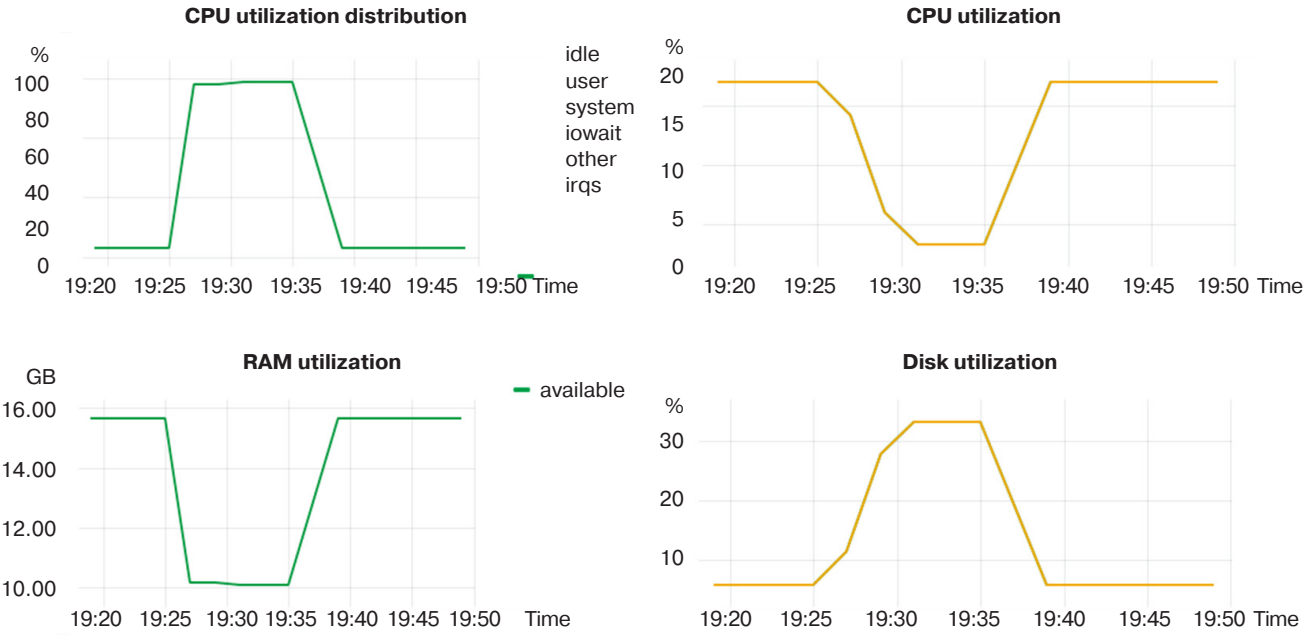

Fig. 11. Load testing with complex queries using temporary tables and 500 connections to the DB filled with test data

Testing demonstrated a significant influence of parallel operations on DB resources along with the possibility to minimize them by means of pool tools.

Let us examine RAM utilization at different number of direct connections to the DB:

1) 100 connections, 1125 MB (11.25 MB/connection);
2) 500 connections, 5900 MB (11.8 MB/connection);
3) 1000 connections, 10250 MB (10.25 MB/connection).

RAM utilization rates when tested via the pooler were extremely low and did not fluctuate depending on the number of connections, remaining at an extremely low level (~250 MB on all tests).

As compared to the previous test, utilization jumps can be observed for all the monitored metrics. The 500-connection test also shows the highest memory utilization per single connection, just like in the previous case. The test for 1000 connections shows

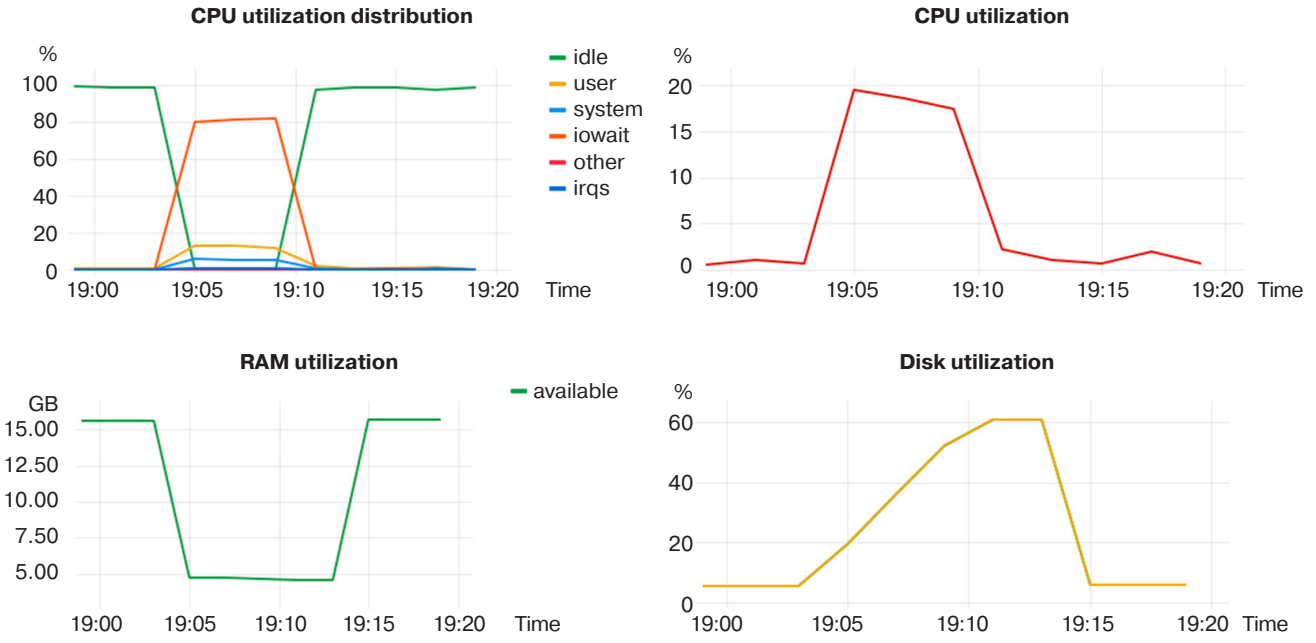

Fig. 12. Load testing with complex queries using temporary tables and 1000 connections to the DB filled with test data

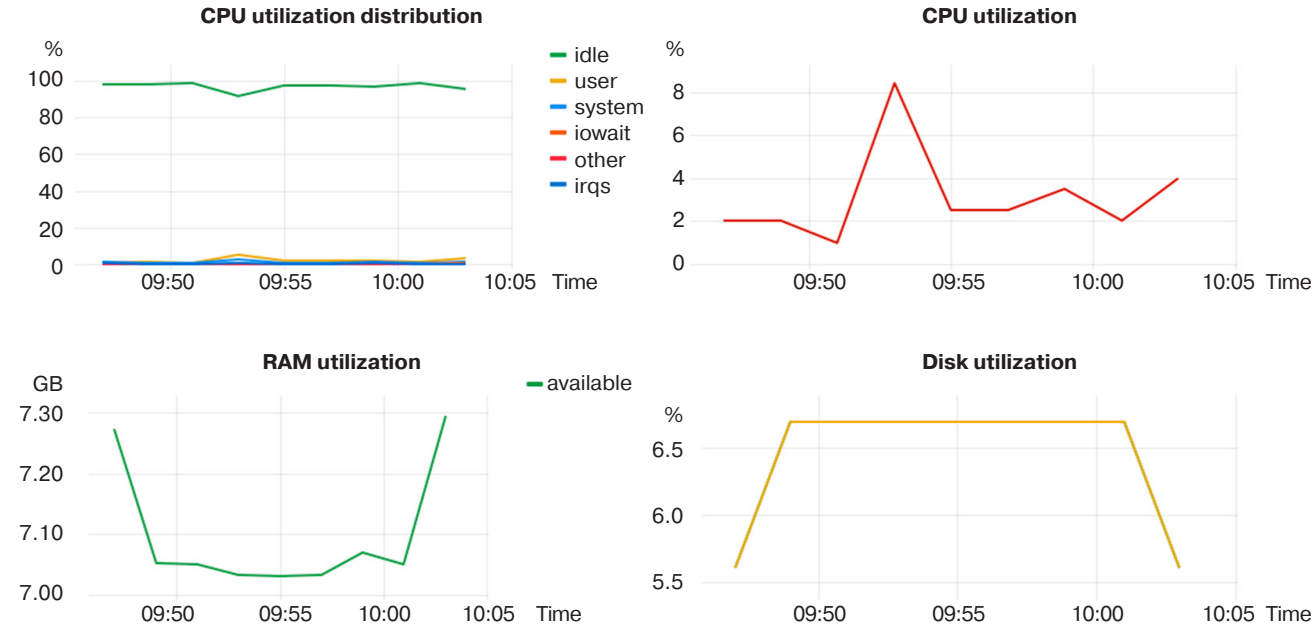

Fig. 13. Load testing with complex queries using temporary tables and 100 connections through the pooler to the DB filled with test data

that the DBMS is not oriented for such a number of connections, since the time of increased CPU and disk system utilization is comparatively higher than the other cases.

Optimization of disk subsystem utilization can also be observed. This is due to the possibility of transferring the processing of operations with temporary tables to the power of the pooler (since it knows the result of all queries in advance).

Reduced CPU load when testing through the pooler is due to the fact that it takes over session management (the most expensive process in terms of CPU resources).

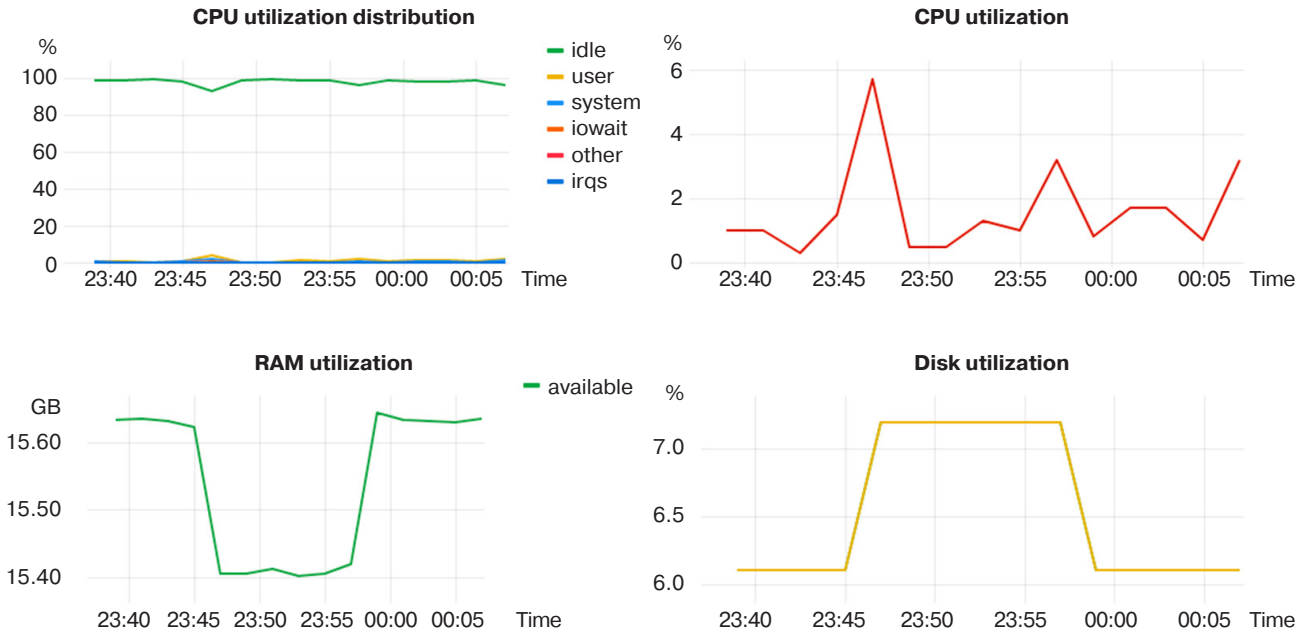

Fig. 14. Performing load testing with complex queries using temporary tables and 500 connections through a pooler to the DB filled with test data

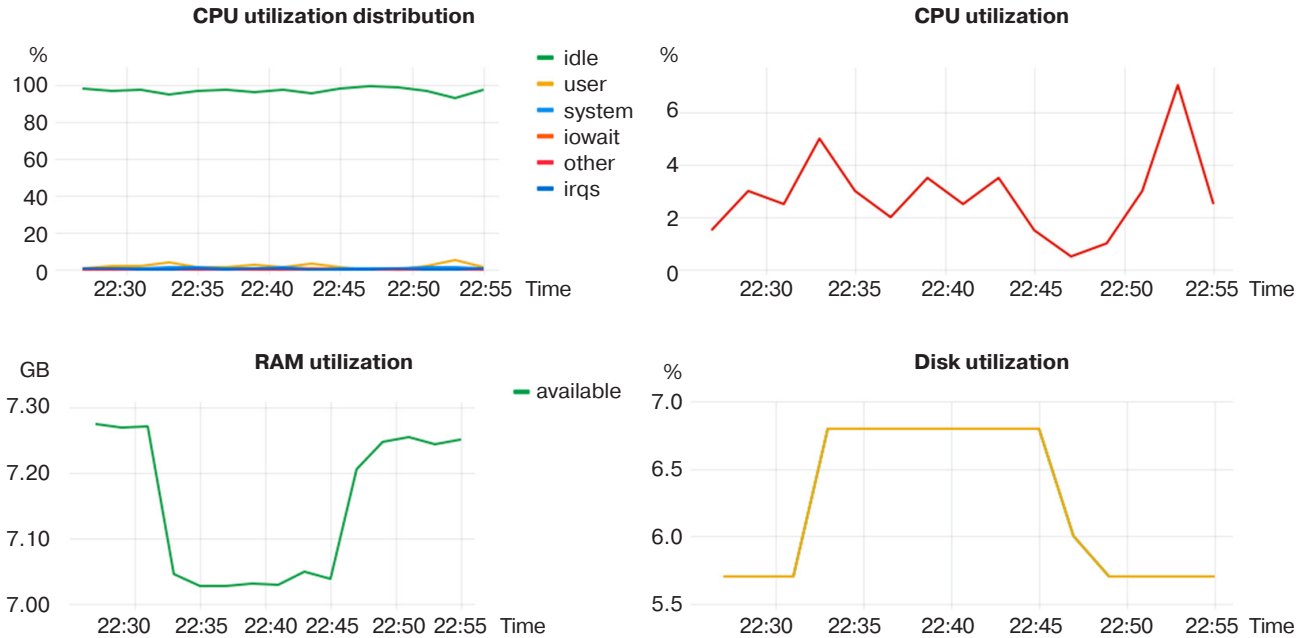

Fig. 15. Load testing with complex queries using temporary tables and 1000 connections through the pooler to the DB filled with test data

3.3. Combined queries

This test, which is a combination of the two previous ones, is aimed at evaluating a more general case of DB usage (parallel writing and reading).

Metrics for direct connections by combined queries are depicted in Fig. 16 (100 connections), Fig. 17 (500 connections), Fig. 18 (1000 connections), where the time segments shown in the graph represent the change in metrics during the testing period in real time in *hours:minutes* format.

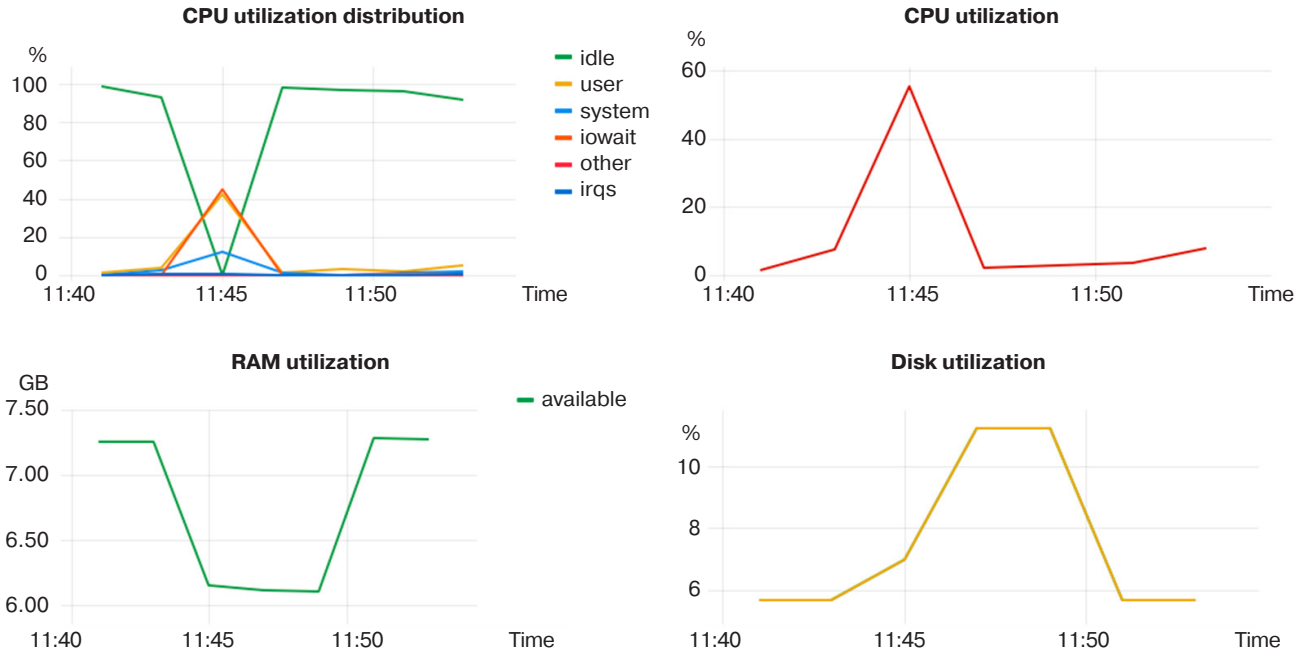

Fig. 16. Performing load testing with combined queries using temporary tables and 100 connections directly to the DB

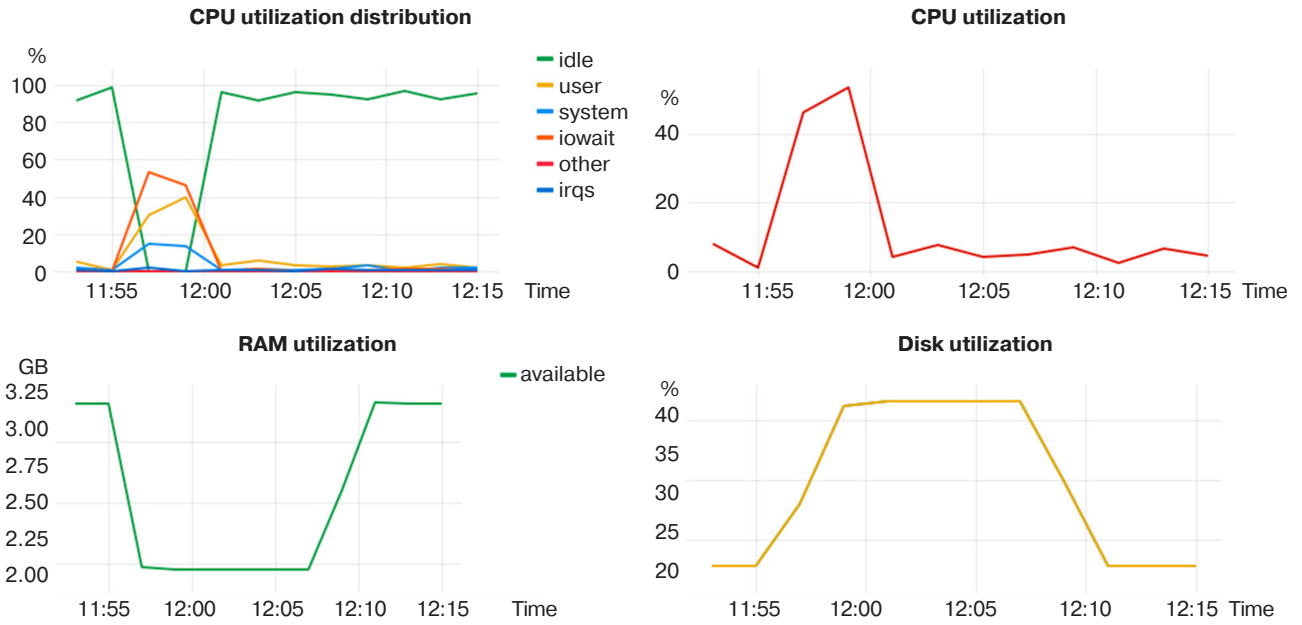

Fig. 17. Performing load testing with combined queries using temporary tables and 500 connections directly to the DB

Metrics for connections via pooler by combined queries are shown in Fig. 19 (100 connections), Fig. 20 (500 connections), and Fig. 21 (1000 connections).

Testing did not show any anomalies when running simultaneous read and write tests. This shows that testing with more real-world cases does not conflict in any way with using the Connection Balancer.

Let us examine RAM utilization at different number of direct connections to the DB:

1) 100 connections, 1200 MB (12.0 MB/connection);
2) 500 connections, 6050 MB (12.1 MB/connection);
3) 1000 connections, 11150 MB (11.15 MB/connection).

RAM utilization rates when tested via pooler were extremely low and did not fluctuate depending on the

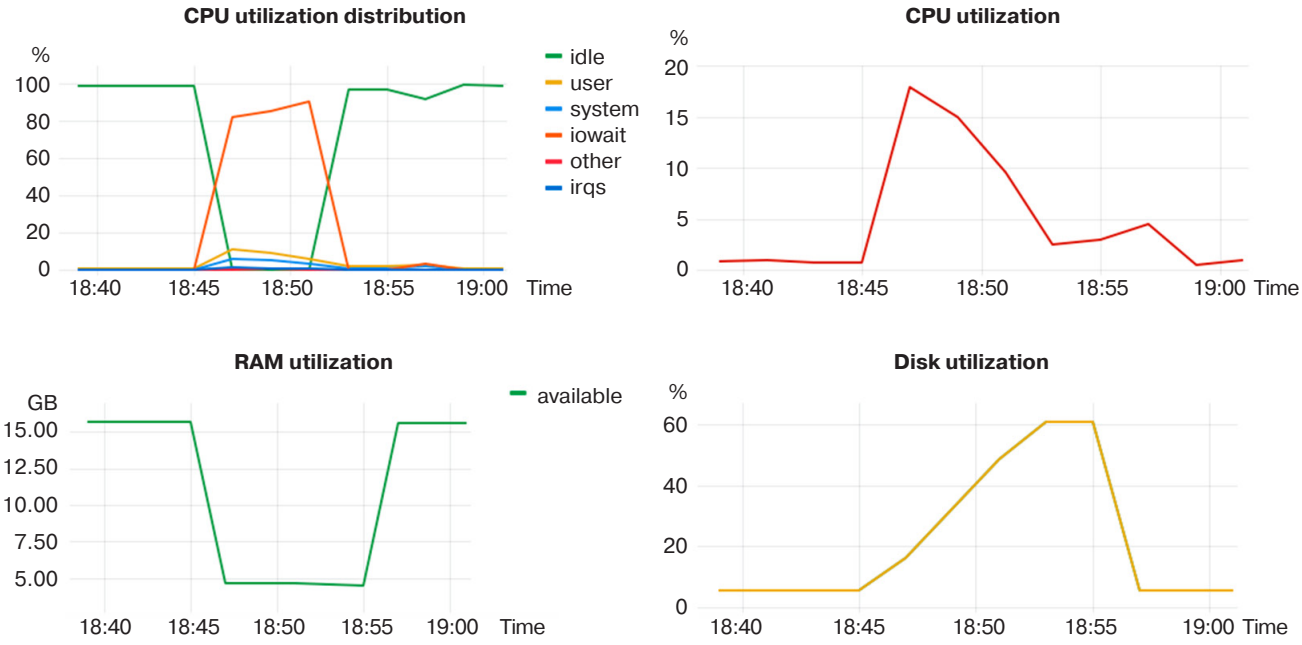

Fig. 18. Performing load testing with combined queries using temporary tables and 1000 connections directly to the DB

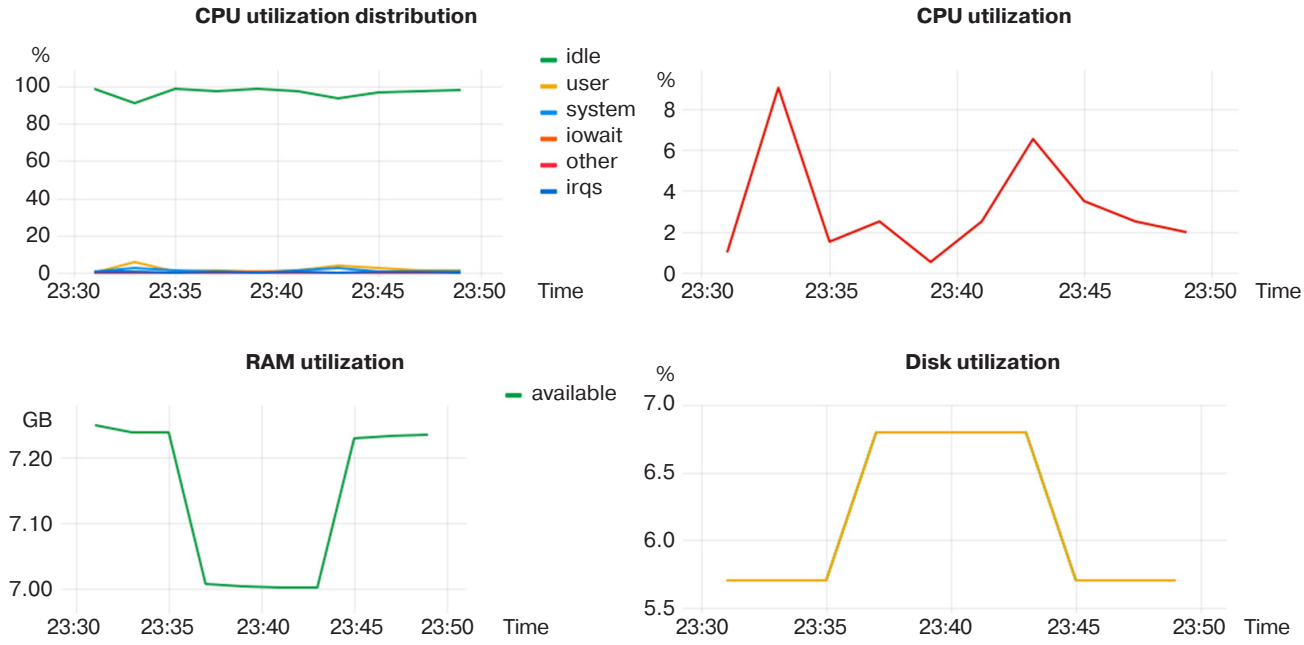

Fig. 19. Performing load testing with combined queries using temporary tables and 100 connections through the pooler to the DB

number of connections, remaining at an extremely low level (~250 MB on all tests).

3.4. Overall assessment of the results

In general, the testing showed the expected results, namely the reduction of the load on the system. Thus, using the balancer will not only reduce infrastructure costs, but also greatly optimize the system itself, especially in terms of avoiding "huge" VM solutions involving several hundred gigabytes of RAM and CPU cores for DBMS. Testing has shown that on average, connection optimization freed 25–50% of RAM that was intended for the DBMS itself, taking into account

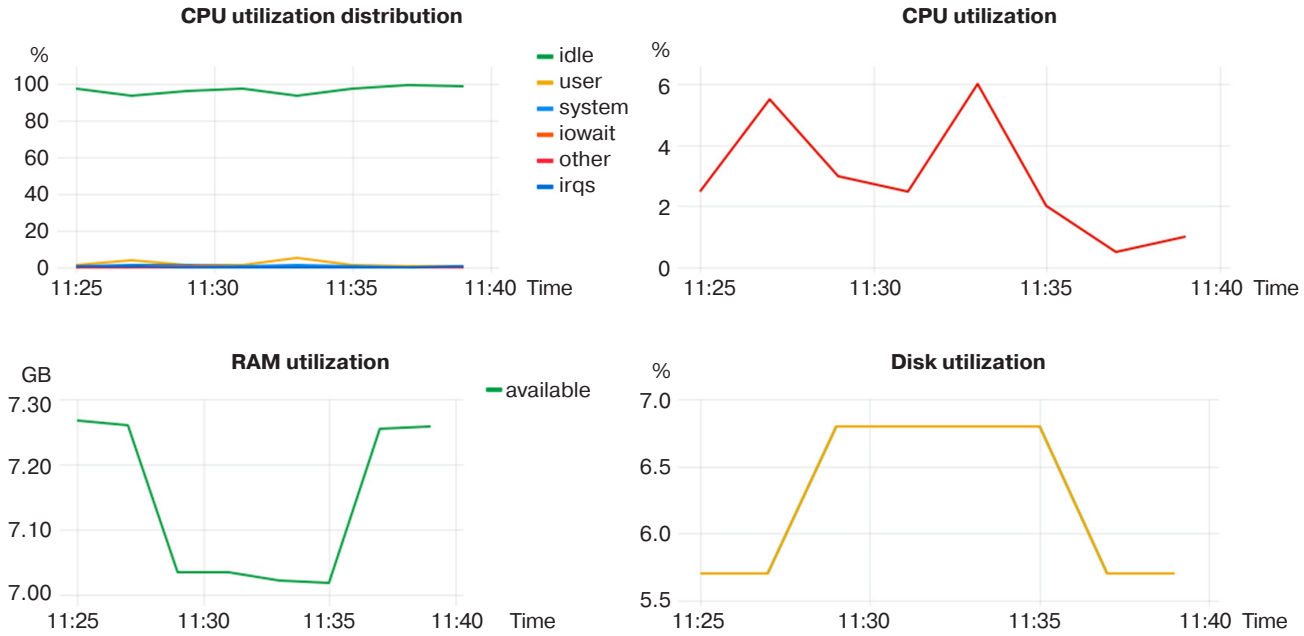

Fig. 20. Performing load testing with combined queries using temporary tables and 500 connections through the pooler to the DB

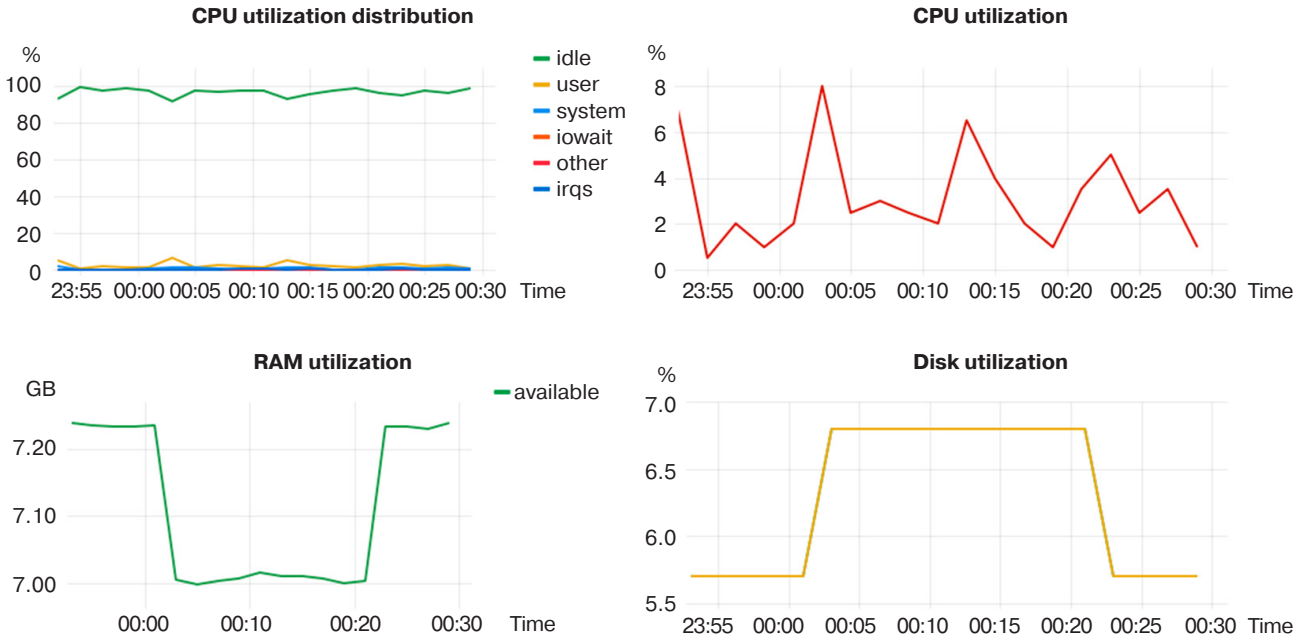

Fig. 21. Performing load testing with combined queries using temporary tables and 1000 connections through the pooler to the DB

Table 2. Results of load testing with simple queries

Number of queries	Method of sending queries to the DB	CPU utilization, %	RAM utilization, MB	Disk utilization, %
100 queries	Directly to the DB	7	230	0
	Through the pooler to the DB	14	30	0
500 queries	Directly to the DB	11	1100	0
	Through the pooler to the DB	8	30	0
1000 queries	Directly to the DB	8	1800	0
	Through the pooler to the DB	11	20	1.4

Table 3. Results of load testing with complex queries

Number of queries	Method of sending queries to the DB	CPU utilization, %	RAM utilization, MB	Disk utilization, %
100 queries	Directly to the DB	25	1125	20
	Through the pooler to the DB	8	260	1.5
500 queries	Directly to the DB	16	5900	30
	Through the pooler to the DB	6	250	1.5
1000 queries	Directly to the DB	19	10250	55
	Through the pooler to the DB	6	250	1.1

Table 4. Results of load testing with combined requests

Number of queries	Method of sending queries to the DB	CPU utilization, %	RAM utilization, MB	Disk utilization, %
100 queries	Directly to the DB	53	1200	7
	Through the pooler to the DB	8	250	1
500 queries	Directly to the DB	58	6050	32
	Through the pooler to the DB	4	260	1
1000 queries	Directly to the DB	18	11150	55
	Through the pooler to the DB	7	250	1

the size of VMs allocated for the pooler itself. If we take pure calculations without taking into account the resources for the balancer itself, the difference amounted to ~30 times. At the same time, *PgBouncer* demonstrated approximately the same values on different tests, which indicates some universality of this solution as opposed to adjusting the DBMS parameters itself.

The reduction of disk subsystem resources utilization when using query balancers should be noted. This optimization will help to reduce costs and decrease the parameters of the disk intended for the DB.

We have also noticed a decrease in CPU utilization when using the pooler. The average fluctuation was 15–20%, which can hardly be called an optimization, since the CPU load has an instantaneous, peak character, and the monitoring system collects data once a minute. For this reason, there can be sharp jumps in readings. The obtained results can be taken into account when designing the system.

"CPU usage distribution" graphs shown in the figures during testing display the information that the resources are directly spent for the *PostgreSQL* DBMS operation and not for other processes, for example, iowait, which occurs at the maximum load on the disk subsystem.

Summarized data of the conducted testing are presented in Tables 2–4.

CONCLUSIONS

The study has demonstrated the effectiveness of *PgBouncer* software for managing a *PostgreSQL* DB connection pool. The testing revealed an improvement in system performance by reducing the resources spent on *PostgreSQL* DBMS, namely, the load on the CPU decreased by 15%, RAM—by 25–50%, and disk subsystem—by 20%.

This balancer has a flexible and easy system of customization of operation modes, allowing the most

suitable option to be selected depending on the specific needs of applications and DB settings.

Using of *PgBouncer* increases DB reliability and reduces query processing time. This is especially important for applications working with a large amount of data that process many queries simultaneously.

Thus, *PgBouncer* can be concluded to represent a useful tool for *PostgreSQL* DB management. It can be successfully applied in many applications and infrastructures where high performance, scalability and security are required.

In the future, it is planned to study ways of deploying this architectural solution on the basis of the Russian platform and to conduct load testing to determine the feasibility of migrating systems, as well as to conduct testing as part of a high-availability cluster.

Authors' contributions

A.S. Boronnikov—the research idea, conducting research, writing the text of the article, interpretation and generalization of the results of the research, scientific editing of the article.

P.S. Tsyngalev—consultations on research issues, writing the text of the article.

V.G. Ilyin—consultations on research issues, writing the text of the article.

T.A. Demenkova—the research idea, research planning, scientific editing of the article.

REFERENCES

1. Sharaev E.V. Using Algorithmic Compositions in PostgreSQL Optimization with Machine Learning Methods. *Nauchnomu Progressu – Tvorchestvo Molodykh*. 2019;3:135–137 (in Russ.).
2. Borodin A., Mirvoda S., Porshnev S., Kulikov I. Optimization of Memory Operations in Generalized Search Trees of PostgreSQL. In: Kozielski S., Mrozek D., Kasprowski P., Małysiak-Mrozek B., Kostrzewa D. (Eds.). Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation. BDAS 2017. *Communications in Computer and Information Science*. 2017;716:224–232. https://doi.org/10.1007/978-3-319-58274-0_19
3. Varakuta P.S., Kozlov R.K. Simulation of the capacity of connection pools to the PostgreSQL database. *Tribuna uchenogo = Tribune of the Scientist*. 2022;5:48–53 (in Russ.).
4. Mukhamedina A., Aidarov A.K. Modern load balancing tools. *Nauchnye issledovaniya 21 veka = Scientific Research of the 21st Century*. 2021;2:105–109 (in Russ.).
5. Gudilin D.S., Zvonarev A.E., Goryachkin B.S., Lychagin D.A. Relational Database Performance Comparation. In: *Proc. 5th International Youth Conference on Radio Electronics, Electrical and Power Engineering* (*REEPE*). March 16–18, 2023. Moscow. https://doi.org/10.1109/REEPE57272.2023.10086872
6. Tupikina M.A. Comparison of database management systems SQLite, MySQL and PostgreSQL. In: *Student Science for the Development of the Information Society: collection of materials of the 8th All-Russian Scientific and Technical Conference. Part 2*; May 22–23, 2018. Stavropol: North Caucasian Federal University; 2018. P. 345–347 (in Russ.).
7. Vinogradova M.V., Barashkova E.S., Berezin I.S., Orelikov M.G., Luzin D.S. An overview of the full-text search system in PostgreSQL post-relational database. *E-SCIO*. 2020;5(44):754–778 (in Russ.).
8. Pantilimonov M.V., Buchatskiy R.A., Zhuykov R.A. Machine code caching in PostgreSQL query JIT-compiler. *Trudy Instituta sistemnogo programmirovaniya RAN = Proceedings of the Institute for System Programming of the RAS* (*Proceedings of ISP RAS*). 2020;32(1):205–220 (in Russ.).
9. Portretov V.S. Comparison of PostgreSQL and MySQL. *Molodezhnaya Nauka v Razvitii Regionov*. 2017;1:136–139 (in Russ.).
10. Chauhan C., Kumar D. *PostgreSQL High Performance Cookbook*. 2nd ed. Birmingham: Packt Publishing; 2017. 360 p.
11. Rogov E.V. *PostgreSQL 15 iznutri* (*PostgreSQL 15 from the Inside*). Moscow: DMK Press; 2023. 662 p. (in Russ.).
12. Novikov B.A., Gorshkova E.A., Grafeeva N.G. *Osnovy tekhnologii baz dannykh* (*Bases of Technologies of Databases*). 2nd ed. Moscow: DMK Press; 2020. 582 p. (in Russ.).
13. Boichenko A.V., Rogojin D.K., Korneev D.G. Algorithm for dynamic scaling relational database in clouds. *Statistika i Ekonomika = Statistics and Economics.* 2014;6–2:461–465 (in Russ.).
14. Afanas'ev G.I., Abulkasimov M.M., Belonogov I.B. How to create a PostgreSQL Docker image on Ubuntu Linux. *Alleya nauki = Alley of Science*. 2018;2(1–17):913–918 (in Russ.).
15. Smolinski M. Impact of storage space configuration on transaction processing performance for relational database in PostgreSQL. In: Kozielski S., Mrozek D., Kasprowski P., Małysiak-Mrozek B., Kostrzewa D. (Eds.). Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation. BDAS 2017. *Communications in Computer and Information Science*. 2018;928:157–167. https://doi.org/10.1007/978-3-319-99987-6_12

СПИСОК ЛИТЕРАТУРЫ

1. Шараев Е.В. Использование алгоритмических композиций при оптимизации PostgreSQL методами машинного обучения. *Научному Прогрессу – Творчество Молодых*. 2019;3:135–137.
2. Borodin A., Mirvoda S., Porshnev S., Kulikov I. Optimization of Memory Operations in Generalized Search Trees of PostgreSQL. In: Kozielski S., Mrozek D., Kasprowski P., Małysiak-Mrozek B., Kostrzewa D. (Eds.). Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation. BDAS 2017. *Communications in Computer and Information Science*. 2017;716:224–232. https://doi.org/10.1007/978-3-319-58274-0_19

3. Варакута П.С., Козлов Р.К. Имитационное моделирование пропускной способности пулов соединений к базе данных PostgreSQL. *Трибуна ученого*. 2022;5:48–53.
4. Мухамедина А., Айдаров А.К. Современные средства балансировки перегрузки. *Научные исследования XXI века*. 2021;2:105–109.
5. Gudilin D.S., Zvonarev A.E., Goryachkin B.S., Lychagin D.A. Relational Database Performance Comparation. In: *Proc. 5th International Youth Conference on Radio Electronics, Electrical and Power Engineering* (*REEPE*). March, 16–18, 2023. Moscow. https://doi.org/10.1109/REEPE57272.2023.10086872
6. Тупикина М.А. Сравнение систем управления базами данных SQLite, MySQL и PostgreSQL. В сб.: *Студенческая наука для развития информационного общества: сборник материалов VIII Всероссийской научно-технической конференции*. Часть 2; 22–23 мая 2018 г. Ставрополь: Северо-Кавказский федеральный университет; 2018. С. 345–347.
7. Виноградова М.В., Барашкова Е.С., Березин И.С., Ореликов М.Г., Лузин Д.С. Обзор системы полнотекстового поиска в постреляционной базе данных PostgreSQL. *E-SCIO*. 2020;5(44):754–778.
8. Пантилимонов М.В., Бучацкий Р.А., Жуйков Р.А. Кэширование машинного кода в динамическом компиляторе SQL-запросов для СУБД PostgreSQL. *Труды Института системного программирования РАН*. 2020;32(1):205–220. https://doi.org/10.15514/ISPRAS-2020-32(1)-11
9. Портретов В.С. Сравнение PostgreSQL и MySQL. *Молодежная наука в развитии регионов*. 2017;1:136–139.
10. Chauhan C., Kumar D. *PostgreSQL High Performance Cookbook*. 2nd ed. Birmingham: Packt Publishing; 2017. 360 p.
11. Рогов Е.В. *PostgreSQL 15 изнутри*. М.: ДМК Пресс; 2023. 662 с.
12. Новиков Б.А., Горшкова Е.А., Графеева Н.Г. *Основы технологий баз данных*. 2-е изд. М.: ДМК Пресс; 2020. 582 с.
13. Бойченко А.В., Рогожин Д.К, Корнеев Д.Г. Алгоритм динамического масштабирования реляционных баз данных в облачных средах. *Статистика и Экономика.* 2014;6–2:461–465.
14. Афанасьев Г.И., Абулкасимов М.М., Белоногов И.Б. Методика создания Docker-образа PostgreSQL в среде Ubuntu Linux. *Аллея науки.* 2018;2(1–17):913–918.
15. Smolinski M. Impact of storage space configuration on transaction processing performance for relational database in PostgreSQL. In: Kozielski S., Mrozek D., Kasprowski P., Małysiak-Mrozek B., Kostrzewa D. (Eds.). Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation. BDAS 2017. *Communications in Computer and Information Science*. 2018;928:157–167. https://doi.org/10.1007/978-3-319-99987-6_12

About the authors

Anton S. Boronnikov, Postgraduate Student, Senior Lecture, Computer Engineering Department, Institute of Information Technologies, MIREA – Russian Technological University (78, Vernadskogo pr., Moscow, 119454 Russia). E-mail: boronnikov-anton@mail.ru. RSCI SPIN-code 8232-6328, https://orcid.org/0009-0008-4911-6609

Pavel S. Tsyngalev, Student, MIREA – Russian Technological University (78, Vernadskogo pr., Moscow, 119454 Russia). E-mail: pstsyngalev@mail.ru. https://orcid.org/0009-0007-6354-1364

Victor G. Ilyin, Student, MIREA – Russian Technological University (78, Vernadskogo pr., Moscow, 119454 Russia). E-mail: vgilyin@yahoo.com. https://orcid.org/0009-0001-0304-3052

Tatiana A. Demenkova, Cand. Sci. (Eng.), Associate Professor, Computer Engineering Department, Institute of Information Technologies, MIREA – Russian Technological University (78, Vernadskogo pr., Moscow, 119454 Russia). E-mail: demenkova@mirea.ru. Scopus Author ID 57192958412, ResearcherID AAB-3937-2020, RSCI SPIN-code 3424-7489, https://orcid.org/0000-0003-3519-6683

Об авторах

Боронников Антон Сергеевич, аспирант, старший преподаватель, кафедра вычислительной техники, Институт информационных технологий, ФГБОУ ВО «МИРЭА – Российский технологический университет» (119454, Россия, Москва, пр-т Вернадского, д. 78). E-mail: boronnikov-anton@mail.ru. SPIN-код РИНЦ 8232-6328, https://orcid.org/0009-0008-4911-6609

Цынгалёв Павел Сергеевич, студент, ФГБОУ ВО «МИРЭА – Российский технологический университет» (119454, Россия, Москва, пр-т Вернадского, д. 78). E-mail: pstsyngalev@mail.ru. https://orcid.org/0009-0007-6354-1364

Ильин Виктор Георгиевич, студент, ФГБОУ ВО «МИРЭА – Российский технологический университет» (119454, Россия, Москва, пр-т Вернадского, д. 78). E-mail: vgilyin@yahoo.com. https://orcid.org/0009-0001-0304-3052

Деменкова Татьяна Александровна, к.т.н., доцент, кафедра вычислительной техники, Институт информационных технологий, ФГБОУ ВО «МИРЭА – Российский технологический университет» (119454, Россия, Москва, пр-т Вернадского, д. 78). E-mail: demenkova@mirea.ru. Scopus Author ID 57192958412, ResearcherID AAB-3937-2020, SPIN-код РИНЦ 3424-7489, https://orcid.org/0000-0003-3519-6683

Translated from Russian into English by Lyudmila O. Bychkova
Edited for English language and spelling by Thomas A. Beavitt