ISSN 2278-3091
Volume 9, No.3, May - June 2020
International Journal of Advanced Trends in Computer Science and Engineering
Available Online at http://www.warse.org/IJATCSE/static/pdf/file/ijatcse204932020.pdf
https://doi.org/10.30534/ijatcse/2020/204932020

Development of an information system for distributed processing of streaming data

Yuri Alexandrovich Kostikov[1], Alexander Mikhailovich Romanenkov[2]
[1]Candidate of Physical and Mathematical Sciences, Head of Department 812, Moscow Aviation Institute (National Research University), Russia, jkostikov@mail.ru
[2]Candidate of Technical Sciences, Associate Professor of Department of 812, Moscow Aviation Institute (National Research University), Russia, romanaleks@gmail.com

ABSTRACT

The paper proposes methods for designing and building information systems for distributed processing of streaming data and their application for organizing and analyzing trade at financial sites. Typical approaches to organizing scalable systems are considered.

Key words: Distributed systems, scalability, server load estimation, load balancing, performance analysis.

1. INTRODUCTION

The paper considers the problem of saving, subsequent processing and analysis of large volumes of data obtained in real time from external data sources. Such data can be generated by thousands of sources and is usually called streaming. For example, they may include information from sensors to monitor performance and prevent possible problems, user actions on the website, information about the state of the financial market, and so on.

The relevance of the work is determined primarily by the fact that more and more serious requirements for performance, scalability and fault tolerance are imposed on large systems in the era of the development of the Internet and information technology. Important ideas and questions were considered in [1-3]. The focus is on the use of cloud technologies for building and maintaining highly loaded systems that operate with large volumes of real-time information. Note that the work [4] shows approaches to the development of a data processing application, which is a monolithic system, a multicomponent system. The purpose of this system implies the presence of moments of peak loads, high requirements for stability and performance. Satisfying all these requirements without using load balancing and decentralized data warehouse methods is a very difficult task. The approaches used in this work can be applied to modernize the architecture of the data processing system.

The aim of the work is the design and development of an information system for distributed processing of streaming data and its use for organizing and analyzing trade on financial sites. The proposed solutions and approaches provide a universal API and organize the receipt and processing of large amounts of information from an arbitrary set of financial trading platforms.

Methodology. This paper describes methods for developing a system for automated processing of streaming data. The concept of big data is used, which is proposed architecture of a distributed information system. The focus is on system availability and scalability. The concept of micro-service architecture that is relevant in the modern world is used for these purposes.

2. DESCRIPTION OF THE MAIN DIRECTIONS DISTRIBUTED SYSTEMS

The problem of choosing between centralized and distributed models for representing computing resources is one of the key problems in the development of computer systems. Today, large systems process millions of events per day; and their number is growing all the time. These systems are required to ensure high fault tolerance and high performance. These limitations mean that almost every major application should be a distributed system.

It is worth noting that one of the key features of distributed systems is that parts of these systems usually consist of various applications that are independent of each other, or of several copies of the same application. The entire system is physically dispersed between many different servers, which can provide scalability and fault tolerance. Interacting with each other, these parts implement a certain service, for example, a highly visited website, various data collection and analysis systems, applications that provide the transmission of multimedia content, etc.

When developing software that requires the use of distributed computing, it is necessary to choose which problems will be solved with their application, and then select the criteria that the system must satisfy.

High Availability is becoming one of the most important tasks in the modern world to ensure the continued availability of the service. The unavailability of the service even for a couple of minutes can lead to missed financial opportunities or even to significant economic losses for the business. Therefore, the development of constantly accessible and fail-safe systems is both a fundamental and

technological requirement. High availability in distributed systems requires consideration of creating redundancy as a reserve for key components, the possibility of quick recovery after partial or complete failures, as well as their minimization and correct processing. Nevertheless, achieving 100% availability is a difficult and expensive task today, so the availability of "four nines" (99.99%, or about 50 minutes of downtime per year) is considered a good level for most systems. However, even this level is not easy to achieve.

Scalability is also one of the serious criteria for building large systems. Scalability refers to increased throughput for handling large volumes of load. It is characterized by various values, for example: the amount of traffic that the system can handle, the ability to increase the total amount of data storage, the number of different operations within the system.

The cost of developing any system is one of the decisive factors. It includes support costs and the infrastructure required to operate. It is worth considering the amount of time spent on development, as well as the level of skill of the developers. In addition, the system should provide the ability to easily diagnose emerging problems, ease of updating, the ability to expand the development team. It should be considered whether data loss or inaccuracy is possible and what percentage is permissible and acceptable.

2.1 Big data and streaming data

The term “big data” refers to various tools, approaches and methods of processing both structured and unstructured data in order to use them later for specific tasks and goals.

The basic principle of big data processing is horizontal scalability. Data is distributed between different servers in a single network, and their processing occurs without performance degradation. The following traditional defining characteristics for big data, developed during the study of Meta Group in 2001, called “3D-V”, are distinguished:

- Data Volume: The amount of physical data volume.
- Data Velocity: growth rate and the need for fast data processing to obtain results.
- Data Variety: The ability to simultaneously process various types of data.

“Streaming data” is information that is constantly generated in real time by small volumes of thousands of sources. The streaming data may include various types of information, for example, statistics, telemetry data received from various devices, user actions, information from financial trading floors. This data should be processed sequentially and incrementally, either for each record, or using a sliding time window, after which it can be used in various analytical tasks, including determining correlation, performing aggregation, filtering, and standardization.

Streaming processing requires the use of two levels: the storage level and the processing level. The storage tier should support write sequencing and strict consistency in order to provide cost-effective, reproducible read and write operations for large amounts of data without sacrificing performance. The processing level is responsible for the consumption of data located at the storage level and notification of the storage level about which data can be archived as currently unclaimed or deleted as unnecessary. In addition, it is necessary to provide scalability, data integrity and fault tolerance both at the storage level and at the processing level.

2.2Cloud technologies

Cloud is a technology that allows convenient network access on demand to some common fund of configurable computing resources (for example, data networks, servers, storage devices, applications and services - both together and separately), which can be promptly provided and released with minimal operating costs or calls to the provider.

Cloud computing consumers can significantly reduce the cost of information technology infrastructure (in the short and medium term) and respond flexibly to changes in computing needs by using the properties of cloud computing elastic computing, that is, dynamically adapt bandwidth by changing the amount of computing resources used for meet the changing workload.

We can conclude from this that the use of resources and solutions provided by cloud providers makes it possible to simplify the development and further support of distributed information systems that meet the requirements for increased performance and fault tolerance.

2.3 Architectural Approaches in Software Development

When designing software designed for work in the cloud infrastructure, the question inevitably arises of choosing the main type of architecture. Due to the widespread development of cloud technologies, several basic approaches are distinguished today: the classic monolithic architecture, building an application based on serverless computing and sharing responsibility between different microservices. In order to understand which approach is most optimal for the task, it is necessary to analyze the features, as well as the advantages and disadvantages of each of them.

2.4Monolithic Architecture

At the moment, when building applications, the simplest and most common approach is the use of a monolithic architecture. Using this approach implies that all application components are designed to work closely with each other using shared resources. With horizontal scaling, such an application is usually completely duplicated on several servers, which imposes some restrictions and increases the cost of operation with increasing load.
This approach is convenient to use for developing small applications when the task is to get a finished product in a

short time. Such an organization of architecture makes it easy to run, test the application on the developer's computer before putting it into operation and apply the standard deployment process. Another important plus is the simplification of data integrity, since monolithic applications often usually use a single database instance. At the same time, complex infrastructure is not required. Usually, either one server or several identical servers with the same configuration are used, while, for example, the database can be deployed in the same place as the application. This also leads to simplified monitoring of the entire system.

A strong connection between the modules is worth noting from the significant disadvantages, respectively, changing one of them can affect the logic of the others. After that, each part of the application should be thoroughly tested, since it is difficult to predict what consequences may be caused even by small changes. In addition, any updates lead to a complete reassembly and redeployment of the entire application, which takes a long time with an increase in the code base. Another serious drawback is that monolithic applications become attached to the initially selected set of technologies due to their nature.

2.5 Using serverless computing

Serverless computing is a fairly new trend that has begun to spread through the development of cloud technology. This approach actually implies the use of a set of functions unrelated to each other, which are executed in response to some event, the call of which is actually carried out by the cloud provider, for example, as a result of receiving an HTTP request. There is no need to deal with infrastructure management issues such as the allocation of the necessary resources to scale and ensure fault tolerance, as well as their maintenance.

The main disadvantage of this approach is that long chains of functions must be built when implementing complex business logic. This, in turn, leads to a more complex architecture and infrastructure management, since it is necessary to ensure the correct interaction between a large number of modules. Also, due to the fact that the execution time of functions is limited, there is no possibility of a continuous subscription to data streams received in real time.

2.6Microservice architecture

The use of microservice architecture has become more and more common in the last few years. This is an approach in which a single application is built as a set of small services, each of which is responsible for a specific task and works independently using its own resources, and, if necessary, interacts with the rest through the API. These services can be written in different languages and use different data storage technologies (see, for example, [1-3]).

The advantages of using microservice architecture when building large systems include simplification of the development and installation of updates, as well as a simpler increase in the number of developers. This ensures that each service is responsible for a limited area of tasks. The main advantages are the possibility of virtually unlimited horizontal scaling with increasing workloads. In addition, the fault tolerance of the system increases overall due to the ability to deploy resources with redundancy. It is worth noting that failures that are not resolved by resource redundancy affect only a small part of the functionality of the entire system.

The disadvantages of this approach are discussed below. Due to the fact that the system is divided into independent parts, the complexity of the organization of interaction between them increases. It also complicates infrastructure management and maintenance. Another disadvantage is the impossibility of ensuring data integrity by means of the DBMS, when it must be controlled by several services at once. However, this problem can be solved, for example, by creating a separate service, which will be the transaction orchestrator.

2.7 Organization of data storage and management

Today, one of the basic needs of business and industry is high-speed data processing in the era of digital transformation. The question of choosing a database management system becomes one of the first steps in the development of most software products. Usually there are two most common areas: relational (SQL) and non-relational (NoSQL). The selection criteria are differences in such matters as flexibility, ensuring data integrity and, importantly, scalability. This requires a fairly thorough analysis of the internal structure of the systems. The principles of operation, the pros and cons of each type, as well as their features are discussed below.

2.8 Relational Database Management System (SQL)

A relational database management system is called a DBMS that manages relational databases, the principle of which is based on the relational model and set theory. The data is an n-ary relation, which, in turn, is represented as a subset of the n-ary Cartesian product of n sets. Each of these relations, i.e., tables, consists of many tuples, i.e., records. The attributes of each record correspond to columns.

The classical scheme of the application with a relational database requires a single server. However, there is a need for scaling with increasing load on it. The most common strategies are discussed below.

The simplest and most common scaling strategy for relational databases is master-slave replication, which creates a complete copy of the database. Thus, there will already be several instead of a single server:

- Master is the main server where all changes (add, update, delete) data occur
- Slave is a secondary server that replicates, that is, copies all data from the master server. It is used to read data and there may be several such servers.

This strategy offloads the main server (master) and transfer read operations to auxiliary ones. In addition, fault tolerance is increased; the application may use another in the event of a failure on one of the servers. If a failure of the primary server is detected, its auxiliary role may take over. Replication is usually supported by the DBMS, and configuration and management are independent of the application. However, it should be noted that this strategy is not a convenient scaling mechanism. The reason lies in the data out of sync and delays when copying from the primary server to the secondary. It is also effective only if the number of read operations prevails over the number of write operations. Replication is most often used to provide fault tolerance.

Another way to scale for relational databases is sharding. Its essence lies in dividing (partitioning or partitioning) the database into separate parts (shards) so that they can be placed on different servers. This process depends on the database schema and, unlike replication, is controlled by the application itself. There are 2 main methods of sharding:

1. Vertical sharding - a table or group of tables is submitted to a separate server. In this case, when developing the application, the corresponding connection for each table must be used.
2. Horizontal sharding - the table is shared between several servers. This approach is usually used if it is necessary to store a very large number of records. Separation occurs according to the following principle:

a. A table with the same data schema is created on several servers;
b. The application defines the condition by which the connection necessary for the operation will be selected;
c. Before each call to the table, the application determines the desired connection and performs operations with the corresponding server.

Sharding is the most effective tool for scaling relational databases, but its use greatly increases the complexity of application development and imposes additional restrictions.

1. The JOIN operator can be used between tables only if they are placed on the same shard.
2. Transactions containing write operations must be performed within the same shard.

2.9Non-relational Database Management System (NoSQL)

The problem of complex organization of effective scaling and maintaining high throughput has led to the use of non-relational database management systems NoSQL ("Not only SQL). NoSQL databases are optimized for applications that require large data operations, which are required to provide high throughput, low latency and flexible data models. All this is achieved by reducing the stringent requirements for data consistency, which are typical, for example, of relational databases. Because of this, most NoSQL systems do not fully meet the ACID criteria, but satisfy the BASE requirements:

1. Basic Availability — Each request is guaranteed to complete.
2. Soft state — the state of the system can change over time, even without entering new data, to achieve data consistency.
3. Eventual consistency - data may not be consistent for some time but come to agreement after some time.

In more detail, various types of non-relational databases, as well as examples and various scenarios of their use are considered. Databases based on the "key value" model support high separability, which allows easy and efficient horizontal scaling. Amazon DynamoDB is an example. This DBMS can provide high throughput for read and write operations of any scale with a delay of not more than a few milliseconds. This performance is ensured by partitioning. The value of the primary key of the record is passed to the internal hash function as input, after which the result determines the section (physical location) in which the record is stored.

Document-oriented databases are a subspecies of a database based on a key-value model. The information in them is usually presented as an object or document in a format similar to JSON or XML. The work is based on document storages, which inside have a tree structure, the leaf nodes of which contain data. When a document is added to the database, information about these nodes is entered into indexes, which makes it possible to efficiently find a data storage location even for a complex organization. An example of such a DBMS is CouchDB, MongoDB, DocumentDB, OrientDB.

Graph databases are based on the use of graph structures as the main data model, which makes it possible to efficiently search and select complex information. An example of a DBMS of this type is JanusGraph. Apache Cassandra, Apache HBase, Google Cloud Bigtable, Oracle BerkeleyDB can serve as a data warehouse for it. For example, using Cassandra, scalability to multiple data centers can be achieved without the need for additional configuration.

Databases that use random access memory as a storage allow processing workloads with low latency, which cannot be achieved using the classical approach with disk storage. The most common DBMS are Redis and Memcached. Amazon DynamoDB Accelerator (DAX) also belongs to this category, which provides up to a tenfold increase in database performance — from milliseconds to microseconds — even when processing several million queries per second.

2. 10 Load balancing

The issue of load planning should be considered at the design stage of any major project. The base here was two works [2, 3]. Initially, the problems of insufficient server performance due to increased workloads can be solved by increasing the server capacity or by optimizing the algorithms used, program code, etc., but there comes a time sooner or later when these measures are insufficient,

and this approach cannot provide increased system fault tolerance. Combining several servers into a single cluster and distributing the load between them using a set of special methods called balancing is one of the methods for solving this problem.

Cluster efficiency directly depends on how the load is distributed between its elements. Such distribution can be carried out using hardware and software tools. The main tool for load balancing in distributed systems is balancing at the network level using algorithms and methods corresponding to the network, transport and application levels of the OSI model. The principles of its work are considered on the example of Amazon AWS Elastic Load Balancing (ELB).

It should be noted that ELB is a distributed system. The balancer is not assigned a specific public IP address at the time of creation. The domain name is allocated instead, for example, MyDomainELB-918273645.us-east-1.elb.amazonaws.com. After which it is necessary to configure a DNS record of the CNAME type indicating that the application domain refers to the domain name allocated for the balancer for the application domain name. The following is an example of request processing using this approach:

1. The client contacts the DNS server to resolve the domain example.com. The DNS server will respond with the name ELB MyDomainELB-918273645.us-east-1.elb.amazonaws.com due to the fact that it is specified as an alias for the domain.
2. The client accesses the DNS server to resolve the name MyDomainELB-918273645.us-east-1.elb.amazonaws.com. DNS records for this domain are controlled by Amazon, because it is located in the amazonaws.com domain. Let the DNS server return, for example, 1.2.3.4.
3. The client opens a connection to the server at the provided IP address 1.2.3.4, which is part of the ELB cluster.
4. The server at 1.2.3.4 proxies a request to one of the EC2 instances from the balancing pool.

The above example uses two stages of scaling. The first of these is performed in step 2, when Amazon's DNS server resolves the ELB domain name to an IP address. At this point, Amazon can distribute traffic across multiple ELB servers by varying the IP addresses assigned to the client. The second stage occurs at step 4, where the selected ELB server proxies a request to one of the EC2 instances located in the ELB pool, after which it is already processed by the application. By changing the size of the balancing pool, it is possible to control the scalability of the application.

Both of these steps are necessary to balance the load with a very large amount of traffic. The second stage allows the application to process a larger number of requests per unit of time than can be achieved using a single instance of EC2: connections are distributed across several servers from the balancing pool, and each of these servers processes only part of their total number. The first stage is necessary in order not to limit the application to the maximum throughput for network traffic, which can be processed by a single server from the ELB.

The load balancing performed in step 2 is implemented using the Round-Robin algorithm to select a specific server from the pool where the code is processed to process the request. The rules specified for the balancer for redirection are taken into account, for example, one group of servers can be used for requests on the path / route1 / *, and another for requests on the / route2 / * path. Also, periodically, the ELB performs a health-check operation to detect server failures in order to ensure that requests are redirected only to those that work correctly. In addition, when using the application balancing type, redirection can be performed based on the contents of the request.

3. GENERAL SYSTEM REQUIREMENTS

During the analysis of various ways of organizing the architecture, it was decided to use a hybrid approach that combines the use of microservices and serverless computing in order to provide maximum flexibility in the development and combine the advantages of each approach.

The final system should be fault-tolerant and well scalable, consist of a set of microservices and serverless functions and provide universal REST and Websocket APIs for receiving data on various currency pairs from financial trading floors, as well as provide the ability to place trading orders and track the status of their execution. Another requirement for the system is that all the necessary parameters for operation must be transferred to microservices and serverless functions using environment variables.

Trading floors do not have a standardized API for working with them and may have differences in the designations of currency pairs; and this is one of the main problems. Based on these requirements, the following groups of microservices were allocated:

– market-key-service is a microservice responsible for adding and storing API keys for authentication on financial sites.
– master-data-service is a microservice that provides storage and editing of the list of available currency pairs, as well as information about conversions between the pair designation in the system and on the trading floor.
– market-data-collector-service is a microservice that collects and provides storage of information about available currency pairs. This information should include the state of the market glass, current quotes and OHLCV data for various time periods, and also provide emulation for filling in the missing.
– market-data-viewer-service is a microservice that makes it possible to receive notifications in a universal format about changes in data on available currency pairs in real time, which are collected by the market-data-collector-service.

– trading-service is a microservice that provides placement and management of trading orders through a universal API. In addition, it should be responsible for recording changes in their status and provide notifications in a universal format.

3.1Selection of core technologies

To reduce the cost of maintaining the system, it was decided to use the Amazon AWS commercial public cloud as a well-proven and proven solution for large technology companies in the infrastructure and platform services market. This cloud is located in several geographically dispersed data centers, which are combined into groups by geographic proximity, called "regions". Several availablilityzone is implemented within each of these regions, which provides increased fault tolerance of hosted services.

Node.js. was chosen as the main platform for implementing the business logic of the system. Due to the wide interest of developers in this platform, its package manager, called NPM, has more than 750,000 modules that can be used in development.

It was decided to use TypeScript as the main programming language for implementing logic inside the system. This language was introduced by Microsoft in 2012 and is positioned as a web application development tool that extends the capabilities of JavaScript. TypeScript is backward compatible with JavaScript and translates to the latter.

To simplify the development process and improve the quality, NestJS progressive Node.js framework was chosen, which makes it possible to create efficient enterprise-level server applications. Using a dependency injection pattern is one of the key concepts in NestJS. This is ensured by the fact that classes such as services, repositories, factories, etc. can be considered as providers. The basic idea is that using them makes it possible to inject dependencies. This means that objects can create different relationships with each other, and the function of constructing and implementing class instances can be largely delegated to the NestJS dependency injection system. Swagger technology was used to document the HTTP structure of the API part, for which NestJS provides a separate module @ nestjs / swagger for automatic generation based on data provided by the developer in the code using decorators.

Ideas for implementing a web server, as well as methods for developing distributed systems and approaches to load balancing are used from sources [1, 3].

3.2 Choosing the best web server

HTTP servers supported by the NestJS framework by default are discussed below. The most common among them in the Node.js ecosystem is Express. Fastify is positioned as one of the fastest web servers for Node.js and was conceptually inspired by the ideas behind Express. To choose the most optimal of them, we compared their performance by benchmarking.

2 EC2 c5.large servers in the Amazon AWS cloud platform were leased for testing. Ubuntu Server 18.04 LTS is installed on each of these servers. The first server is used as a host to load, Node.js 10.15.3 LTS is additionally installed on it. Express 4.16.4 and Fastify 1.13.3 packages are used for testing. The wrk benchmarking tool is used for the server that provides the load.

For testing, we prepared data in JSON format, consisting of 10 elements, to emulate the operation of an HTTP server in real mode, and also developed a program using the Express web server and a program using the Fastify web server. 2 preliminary runs without recording the data obtained are performed before each stage of testing to increase reliability. During testing, 3 endpoints of different types (GET, POST) with different payload sizes were used. Each of these tests was carried out using 10, 100, 500, 1000, 2000 and 4000 parallel open connections. The duration of each test was 2 minutes. Listing 1 shows an example query for testing the Express web server.

```
const express = require("express");
constbodyParser = require("body-parser");
const DATA = require("./data");
const app = express();
app.use(bodyParser.json());
app.get("/data", (req, res, next) =>res.send(DATA));
app.get("/data/one", (req, res, next)
=>res.send(DATA[0]));
app.post("/data", (req, res, next) =>
res.send({ success: true, title: req.body.title })
);
app.listen(80, () => console.info(`Express server
running`));
```

Listing 1.Testing Express.

To test the Fastify web server, we used almost the same code as in Listing 1, so we did not list it here.
The test results are presented in tables 1 - 3 and in figures 1 - 3.

Table 1: Test results for a GET request (1 element)

Numberofconcurrentconnections	Express (requests/ s)	Fastify (requests/ s)	Difference(%)
10	1211.23	1493.6	18.9
100	1701.91	2537.89	32.94
500	2270.75	2589.23	12.3
1000	1892.02	2241.74	15.6
2000	1811.45	2084.52	13.1
4000	1798	2162.25	16.8

Table 2: Test results for a GET request (10 elements)

Numberofconcurrentconnections	Express (requests/ s)	Fastify (requests/ s)	Difference (%)
10	2257.35	2356.32	4.2
100	2318.66	2843.24	18.45
500	2486.4	3021.14	17.7
1000	1818.55	2281.74	20.3
2000	1689.6	2114.64	20.1
4000	1714.9	2152.78	20.34

Table 3: Test results for POST request (1 element)

Numberofconcurrentconnections	Express (requests/ s)	Fastify (requests/ s)	Difference(%)
10	3190.73	3905.42	18.3
100	3035.94	3862.52	21.4
500	2576.21	3794.12	32.1
1000	1569.66	1952.31	19.6
2000	1413.93	1785.27	20.8
4000	1789.97	1896.15	5.6

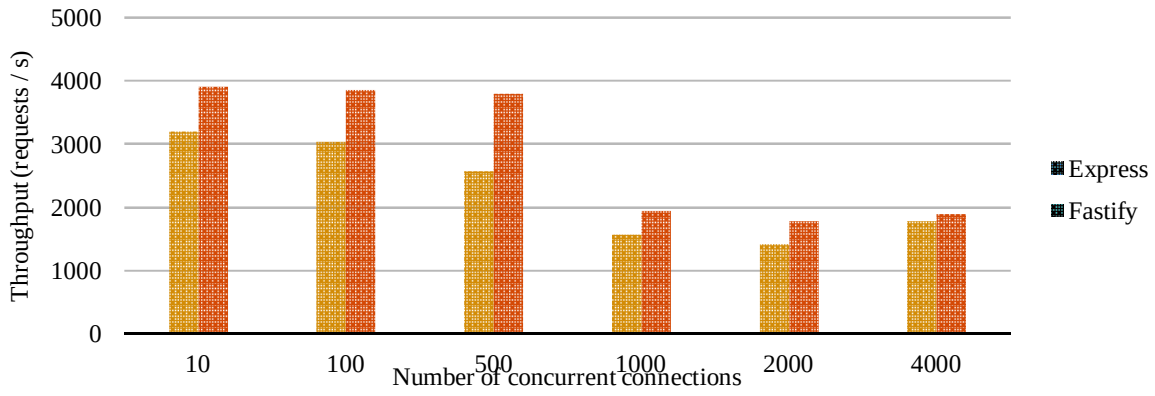

Figure 1: Graph of test results for a GET request (1 element)

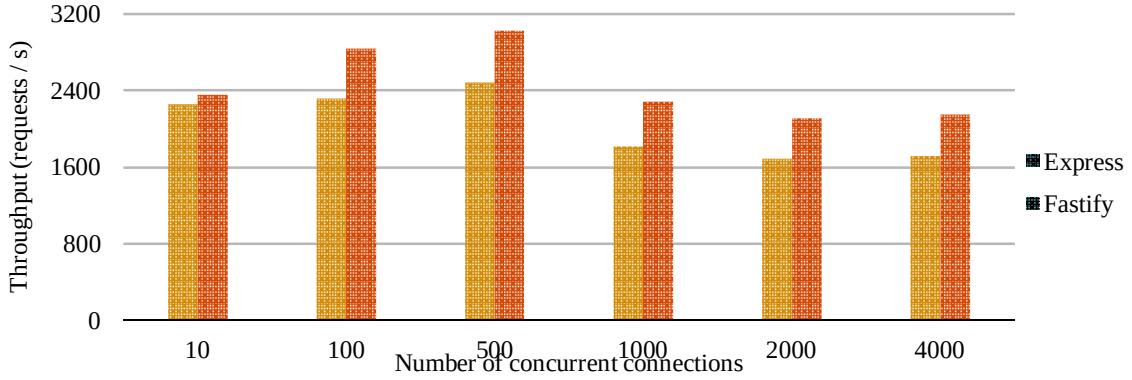

Figure 2: Graph of test results for POST request (1 element)

Based on the data obtained, it can be concluded that Fastify exceeds Express in terms of the number of simultaneously processed requests by about 20%. The use of such specific packages as Passport, which are incompatible with Fastify, was not required during operation; therefore, preference is given to him as the main HTTP web server in order to increase the overall system performance.

3.3 DBMS selection

Based on the information received, an assumption is made that it is preferable to use NoSQL to ensure high throughput and fault tolerance at the DBMS level. We tested this assumption using the Yahoo Cloud System Benchmark Testing System (YCSB). This system makes it possible to test the DBMS for simple operations, such as reading, writing and updating, and to obtain indicators of throughput and response time on various system loadings to study its performance. PostgreSQL relational DBMS as the easiest to scale and 2 non-relational: Apache Cassandra and Amazon DynamoDB was chosen for testing.

We rented 3 EC2 c5.2xlarge servers in the Amazon AWS cloud platform for testing. The Ubuntu Server 18.04 LTS operating system is installed on each of these servers. PostgreSQL 11.2 DBMS is installed on the first server, Apache Cassandra 3.11.4 is installed on the second server. The third server is used as providing load, it runs YCSB. A configuration with 100 units of read resources and 100 units of write resources was chosen for testing Amazon DynamoDB, which roughly corresponds to the price of 1 EC2 c5.2xlarge server. The results obtained during testing are shown in Figures 5-10.

Analyzing the test results, we can conclude that the assumption of higher performance for read and write operations for NoSQL DBMS is correct. Among the tested NoSQL solutions, Amazon DynamoDB is the most productive, which is also confirmed by testing conducted as part of the Gagarin Readings XLV scientific conference. As a result, Amazon DynamoDB was chosen as the main database for data that requires long-term storage.

4. SYSTEM ARCHITECTURE DEVELOPMENT

It was decided to use Docker containerization technology in order to automate the assembly process, simplify deployment and increase its reliability due to reproducibility of the environment. This technology “packs” the application with all its environment and dependencies into a container, which can be ported to any Linux system with cgroups support in the kernel, and also provides an environment for managing containers.

A special script file called the Dockerfile is used to describe containerization rules. Note that starting with version 17.05, Docker began to support multi-stage builds, which allow maintaining the minimum size of output containers.

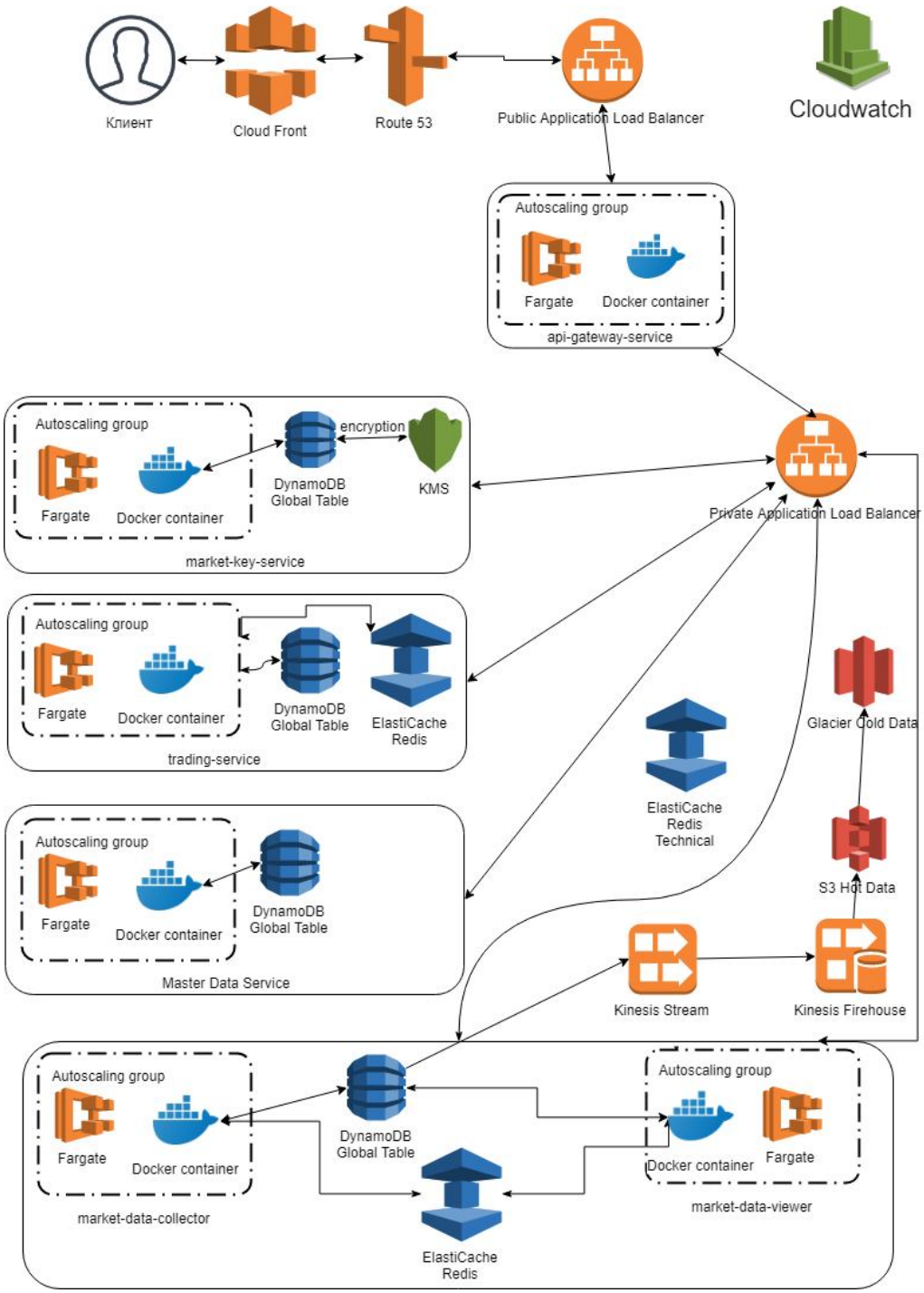

Figure 3: Scheme of the overall system architecture

The universal Dockerfile (shown in Listing 2) for microservices with support for multi-stage assembly, which was obtained during the work, is discussed below. Each of the stages in it is based on the basic Linux Alpine image, which makes it possible to reduce the time required to download the dependencies, complete the assembly and the size of the final container due to the fact that this image contains only the minimum necessary set of tools. The following is an overview of each of the steps:

1. The builder_dependencies stage is the loading of the dependencies that are required to complete the assembly and operation of the application.
2. The production_dependencies stage is similar to stage 1, but only those dependencies that are directly needed for the application to work are loaded.
3. Stage builder copies the dependencies from stage 1 and builds the application.

It is worth noting that for stages 1 - 2, a private key is additionally transferred to access repositories with dependencies if some of the required for the application are located in the git repository, and not in the NPM registry.

After performing steps 1 - 3, the dependencies necessary for the application to work from stage 2 are copied and the assembled application is copied from stage 3, after which the run rules are set. This approach ensures the smallest possible size of the resulting container and does not reinstall dependencies if their list has not changed.

```
FROM node:10-alpine asbuilder_dependencies
ENV NODE_ENV build
RUNapk update &&apk upgrade && \
apk add --no-cache gitopenssh-client
USER node
WORKDIR /home/node
ARGssh_private_key
RUNmkdir .ssh&& \
echo"$ssh_private_key"> ~/.ssh/id_rsa&& \
chmod 600 ~/.ssh/id_rsa&& \
echo"Host *">> ~/.ssh/config&& \
echo" StrictHostKeyChecking no">> ~/.ssh/config&& \
echo" UserKnownHostsFile=/dev/null">> ~/.ssh/config
COPY ./package.json /home/node/
COPY ./yarn.lock /home/node/
RUNyarn install --pure-lockfile
#
FROM node:10-alpine asproduction_dependencies
ENV NODE_ENV production
RUNapk update &&apk upgrade && \
apk add --no-cache gitopenssh-client
USER node
WORKDIR /home/node
ARGssh_private_key
RUNmkdir .ssh&& \
echo"$ssh_private_key"> ~/.ssh/id_rsa&& \
chmod 600 ~/.ssh/id_rsa&& \
echo"Host *">> ~/.ssh/config&& \
echo" StrictHostKeyChecking no">> ~/.ssh/config&& \
echo" UserKnownHostsFile=/dev/null">> ~/.ssh/config
COPY ./package.json /home/node/
COPY ./yarn.lock /home/node/
RUNyarn install --pure-lockfile
# ---
FROM node:10-alpine as builder
ENV NODE_ENV build
USER node
WORKDIR /home/node
COPY --from=builder_dependencies
/home/node/node_modules/ /home/node/node_modules/
COPY . /home/node
RUN yarn run tsc:build
# ---
FROM node:10-alpine
ENV NODE_ENV production
USER node
WORKDIR /home/node
COPY --from=builder /home/node/.env.example
/home/node/
COPY --from=builder /home/node/dist/ /home/node/dist/
COPY --from=production_dependencies
/home/node/node_modules/ /home/node/node_modules/
CMD ["node", "dist/main.js"]
```

Listing 2. Generic Dockerfile code

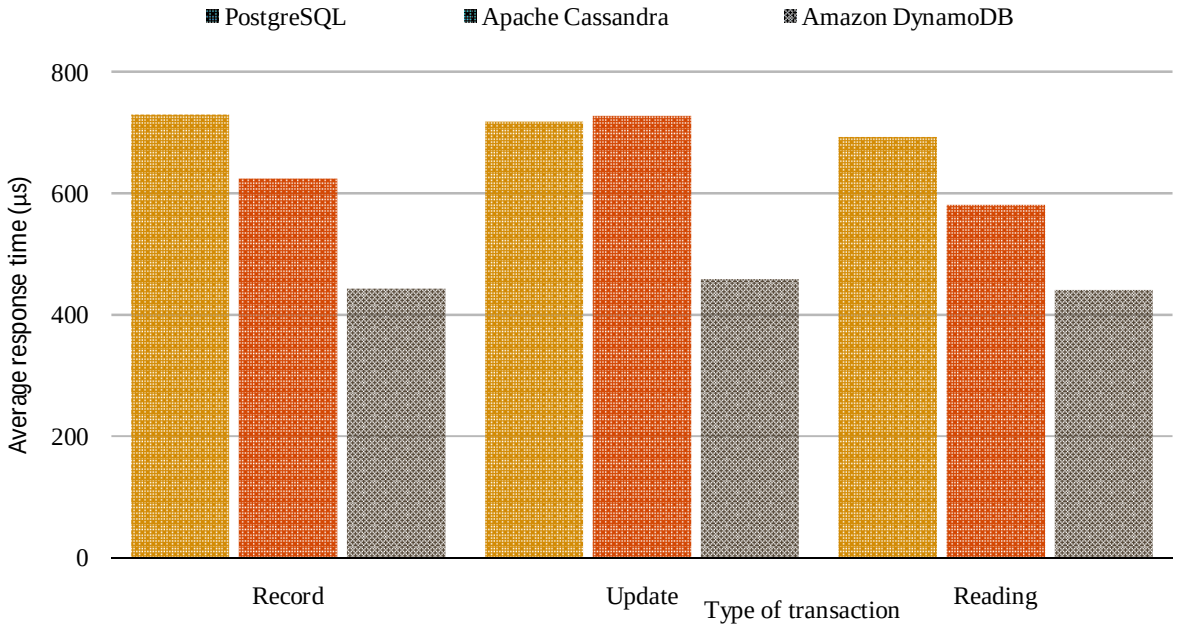

Figure 5: Graph of average response time for a small amount of data

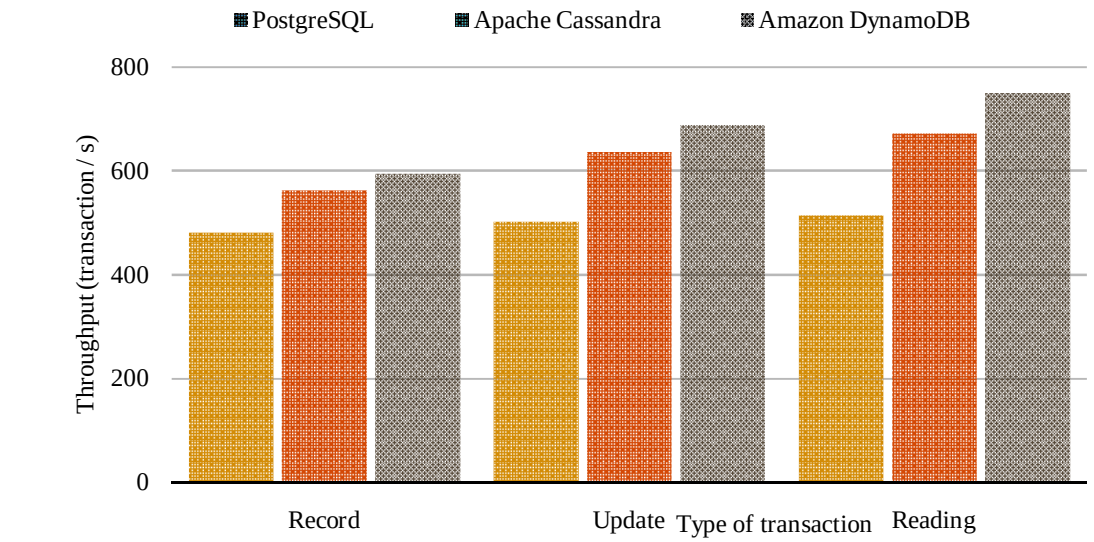

Figure 6: Bandwidth graph for small data volumes

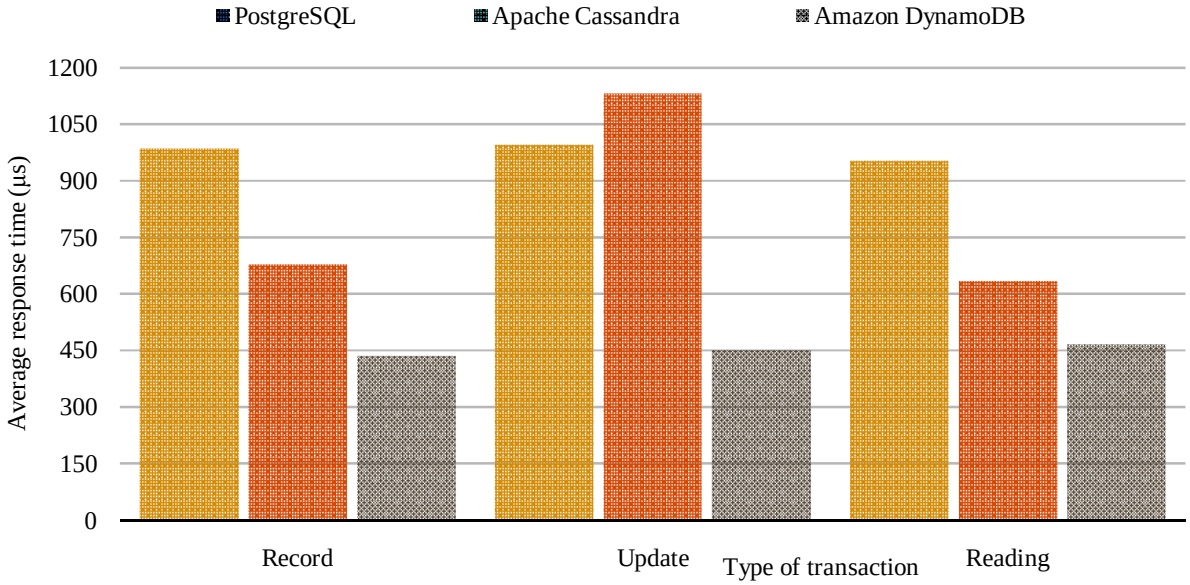

Figure 7: Average response time graph for average data volume

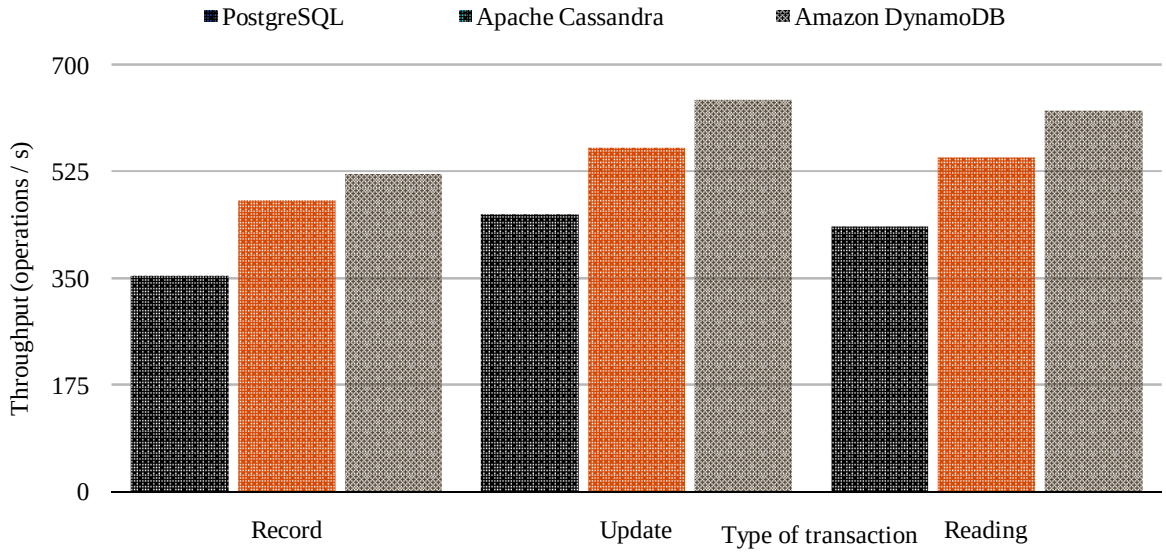

Figure 8: Bandwidth graph for average data volume

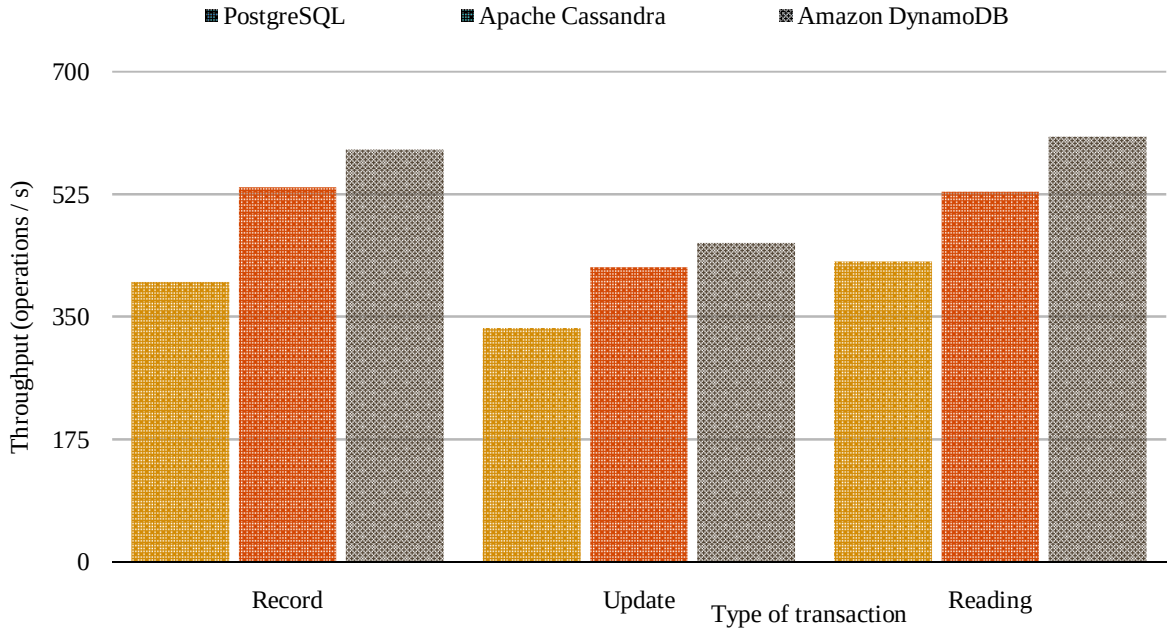

Figure 9: Graph of average response time for a large amount of data

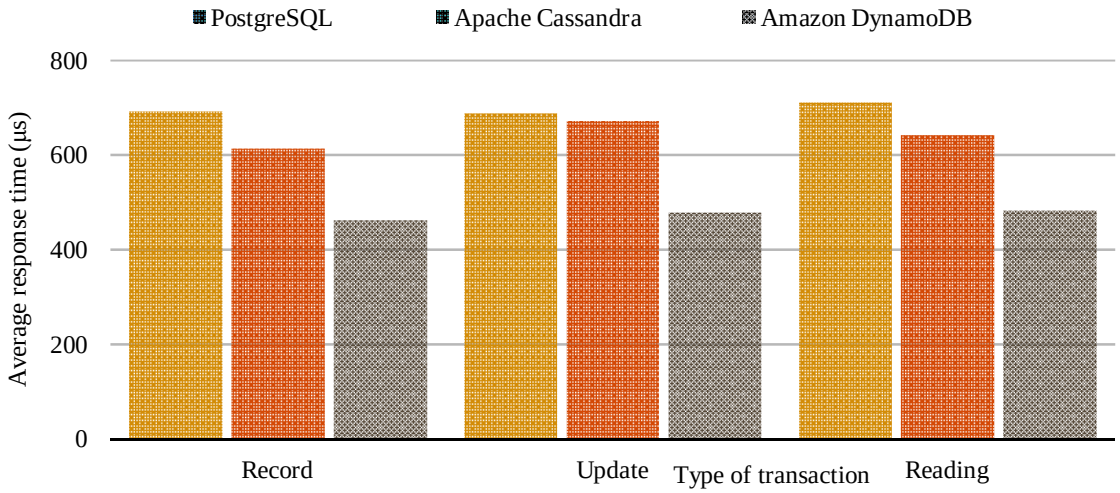

Figure 10: Bandwidth graph for a large amount of data

5. CONCLUSION

The paper considers the task of designing information systems for processing streaming data. A comparative analysis of the methods and approaches that are used to build such systems is carried out. The architecture of a distributed system that provides horizontal scaling is proposed, the results of test launches are shown, and time diagrams of the average response time and throughput of the medium and large data arrays are presented. The results determine the most effective data processing strategy and technology, depending on its size.

REFERENCES

1. V. Cardellini. **A performance study of distributed architectures for the quality of web services**, in*Proceedings of the 34th Conference on System Sciences*, 2001, Vol.10, pp. 213-217.
2. E. Casalicchio,and M.Colajanni. **A client aware dispatching algorithm for web clusters providing multiple services**, in*Proceeding of the 10th International Conference on WWW*,2001,pp. 535-544. https://doi.org/10.1145/371920.372155
3. Valeria Cardellini, Michele Colajanni, and Philip S.Yu. **Dynamic Load Balancing on Web-server Systems**, in*IEEE Internet Computing*, 1999, Vol.3, No. 3, pp. 28-39. https://doi.org/10.1109/4236.769420
4. Yu.A. Kostikov, V.Yu. Pavlov, A.M. Romanenkov, and V.B. Ternovskov. **Adaptive architecture of a hardware-software complex for data storage and processing**,in *Economics: yesterday, today, tomorrow*, 2017, Vol. 7,No. 9A,pp. 192-207.