1. The model describes a system of physical objects (corresponding to the vertices and edges of the graph).
2. The algorithm (approximately) calculates a state of equilibrium for this system.

Description of the model is based on ideas about what can be considered a good image in each case. With the model associated objective function describing a concrete concept of good images, and the algorithm used to optimize the objective function.

The same method of simulated annealing and genetic algorithms are universal approaches to optimize any of the criteria of quality, and are used for all classes of graphs. Among their advantages also include ease of implementation and clarity for the user.

Thus, the following conclusions can be made. Firstly, the algorithms to automatically place graphs give good results only in certain classes of them. Secondly, the image produced by the graph strongly depends on the particular application. We must fulfill certain quality criteria that apply image. Not all graph visualization algorithms can ensure the implementation of the criteria. Therefore, an important finding of such approaches that would allow not only to meet certain quality criteria, but also the opportunity to work with arbitrary graphs.

REFERENCES

1. Конечная математика // Большая советская энциклопедия : в 30 т. / гл. ред. А.М. Прохоров. – 3-е изд. – М. : Сов. энцикл., 1969–1978.
2. Дискретная математика и комбинаторика / под. ред. Джеймс А. Андерсон ; пер. с англ. – М. : Вильямс, 2004. – 960 с.
3. Асанов, М.О. Дискретная математика: Графы, алгоритмы : учеб. пособие / М.О. Асанов, В.А. Баранский, В.В. Расин / под. ред. М.О. Асанов. – 2-е изд. – М. : Лань, 2010. – 368 с.
4. Визуализация графов [Электронный ресурс] / Википедия – свободная энциклопедия. – Режим доступа: http://ru.wikipedia.org/wiki/Визуализация_графов. – Дата доступа: 01.01.2016.
5. Графы в программировании: обработка, визуализация и применение / под. ред. В. Касьянова, В. Евстигнеева. – СПб., 2003. – 1104 с.

UDC 004.75

APPROACHES TO THE CONSTRUCTION OF DISTRIBUTED WEB SYSTEMS

YURI LAPTEV, RYKHARD BOHUSH
Polotsk State University, Belarus

This paper dwells on some of the key issues that should be considered in the design of large websites, as well as some of the basic components used to achieve these goals. The main attention is paid to the analysis of web-based systems.

Open source software has become a fundamental building block for some of the biggest websites. Building and operating a scalable web site at a primitive level is just connecting users with remote resources via the Internet — the part that makes it scalable is that the resources, or access to those resources, are distributed across multiple servers. The time to plan ahead when building a web service can help in the long run. Below are some of the key principles that influence the design of large-scale web systems: performance, cost, reliability, availability, scalability and manageability [1].

The speed of a website affects its usage and user satisfaction, as well as search engine rankings. As a result, a system that is optimized for fast responses is created. A system needs to be reliable, so that a request for data will consistently return the same data. In case if the data changes or is updated, the same request should return the new data. Users should be sure that no data will be lost. Designing systems to be available to failure is a fundamental and a technology requirement. High availability in distributed systems requires careful consideration of redundancy for key components, rapid recovery in case of partial system failures. For some of the larger online retail sites, being unavailable for even minutes can result in thousands or millions of dollars in lost revenue. The effort required to increase capacity to handle greater amounts of load, commonly referred to as the scalability of the system, is very important. Scalability can refer to many different parameters of the system: how easy it is to add more storage capacity, or even how many transactions can be processed. Designing a system that is easy to operate is another important consideration. Things to consider for manageability are the ease of diagnosing and understanding problems when they occur, the ease of making updates or modifications [2].

Each of these principles provides the basis for decisions in designing a distributed web architecture. However, they can also be at odds with one another, so that achieving one objective comes at the cost of another.

When designing any sort of web application it is important to consider these key principles to find the most suitable solution for each specific task.

When it comes to system architecture, there are a few things to consider: what the right pieces are, how these pieces fit together, and what the right tradeoffs are. The initial system planning can save substantial time and resources in the future. Consider the example of hosting for downloading images. Imagine a system where users are able to upload their images to a central server, and the images can be requested via a web link. There is no limit to the number of images that will be stored, necessary data reliability, as well as the need to be fast loading operations and query images. In addition, the system should be easy to maintain and cost effective.

When considering scalable system, it helps to decouple functionality and think about each part of the system as its own service with a clearly defined interface. Each service has its own distinct functional context, and interaction with anything outside of that context takes place through an abstract interface. In a simple design of the architecture all requests to upload and retrieve images are processed by the same server, however, as the system needs to scale it makes sense to break out these two functions into their own services. Another potential problem with this design is that a web server typically has an upper limit on the number of simultaneous connections it can maintain. Since reads can be asynchronous, or take advantage of other performance optimizations like gzip compression or chunked transfer encoding, the web server can switch serve reads faster and switch between clients quickly serving more requests per second than the max number of connections. Writes, on the other hand, tend to maintain an open connection for the duration for the upload, with the result that the Web server cannot handle the new clients until other processes of recording information complete them. Planning for this sort of problem makes a good case to split out reads and writes of images into their own services. This allows us to scale each of them independently, but also helps clarify what is going on at each point. Finally, this separates future concerns, which would make it easier to troubleshoot and scale a problem like slow reads [3].

In order to handle a failure, web architecture should have redundancy of its services and data. For example, if there is only one copy of a file stored on a single server, then losing that server means losing that file. A common way of handling it is to create multiple, or redundant, copies. This same principle also applies to services. Creating redundancy in a system can remove single points of failure and provide a backup or spare functionality if needed in a crisis. For example, if there are two instances of the same service running in production, and one fails or degrades, the system can *failover* to the healthy copy. Another key part of service redundancy is creating a *shared-nothing architecture*. With this architecture, each node is able to operate independently of one another. This helps a lot with scalability since new nodes can be added without special conditions or knowledge. There is no single point of failure in these systems, so they are much more resilient to failure [4].

There may be very large data sets that are unable to fit on a single server. It may also be the case that an operation requires too many computing resources. In either case, you have two choices: scale vertically or horizontally. Scaling vertically means adding more resources to an individual server. Therefore, for a very large data set, this might mean adding more hard drives so a single server can contain the entire data set. In the case of the compute operation, this could mean moving the computation to a bigger server with a faster CPU or more memory. To scale horizontally, on the other hand, is to add more nodes. In the case of the large data set, this might be a second server to store parts of the data set, and for the computing resource, it would mean splitting the operation or load across some additional nodes. In our image server example, it is possible that the single file server, used to store images, could be replaced by multiple file servers, each containing its own unique set of images. Such an architecture would allow the system to fill each file server with images, adding additional servers as the disks become full.

Of course, there are challenges distributing data or functionality across multiple servers. One of the key issues is data locality; in distributed systems the closer the data to the operation or point of computation is, the better the performance of the system becomes. Therefore it is potentially problematic to have data spread across multiple servers, as any time it is needed it may not be local, forcing the servers to perform a costly fetch of the required information across the network. Another potential issue comes when there are different services reading and writing from a shared resource. For example, if one client sent a request to update image with a new title, but at the same time another client was reading the image. Under these circumstances, it is unclear which title would be the one received by the second client.

We now consider the question of access to data scaling. Most simple web applications run on the following principle: the user through the Internet refers to the application server, and that in turn communicates with the database server. As they grow, there are two main challenges: scaling access to the app server and to the database. Most systems can be simplified to the form when the user directly access the data. For the sake of this section, let us assume you have many terabytes (TB) of data and you want to allow users to access small portions of that data at random. Get access to specific data is particularly difficult because the loading large amounts of data in memory can be very note and directly affects the amount of disk IO. Moreover, even with unique IDs, solving the problem of knowing where to find that little bit of data can be an arduous task. Thankfully, there are many options that you can employ to make this easier: four of the more important ones are caches, proxies, indexes and load balancers.

Caches take advantage of the locality of reference principle: recently requested data is likely to be requested again. They are used in almost every layer of computing: hardware, operating systems, web browsers, web applications and more. A cache is like short-term memory: it has a limited amount of space, but is typically faster than the original data source and contains the most recently accessed items. Caches can exist at all levels in architecture, but are often found at the level nearest to the front end, where they are implemented to return data quickly without taxing downstream levels.

At a basic level, a proxy server is an intermediate piece of hardware/software that receives requests from clients and relays them to the backend origin servers. Typically, proxies are used to filter requests, log requests. Proxies are also immensely helpful when coordinating requests from multiple servers, providing opportunities to optimize request traffic from a system-wide perspective. One way to use a proxy to speed up data access is to collapse the same (or similar) requests together into one request, and then return the single result to the requesting clients.

Using an index to access your data quickly is a well-known strategy for optimizing data access performance; probably the most well-known when it comes to databases. An index makes the trade-offs of increased storage overhead and slower writes for the benefit of faster reads.

Load balancers are a principal part of any architecture, as their role is to distribute load across a set of nodes responsible for servicing requests. Their main purpose is to handle a lot of simultaneous connections and route those connections to one of the request nodes, allowing the system to scale to service more requests by just adding nodes. In a distributed system, load balancers are often found at the very front of the system, such that all incoming requests are routed accordingly. Load balancers also provide the critical function of being able to test the health of a node, such that if a node is unresponsive or over-loaded, it can be removed from the pool handling requests, taking advantage of the redundancy of different nodes in your system [5].

The development of effective systems with fast access to large amounts of data is a very interesting topic, and there are many different approaches that give the right to form the architecture of systems in the early stages of development. When designing a distributed web-based systems there are always a number of difficulties, the solution of which will have to sacrifice some principles to make full use of others. Some useful ways to develop a scalable system are: the separation of functionality to services, the use of redundancy to address failures, the use of data partitioning. With the growth of applications often use to simplify the systems approach, the main ones are proxy, indexes, caches and load balancers.

REFERENCES

1. Abbott, Martin L. Scalability Rules: 50 Principles for Scalling Web Sites / Martin L. Abbott. – 1st Edition. – Addison-Wesley, 2011.
2. Abbott, Martin L. The Art of Scalability: Scalable Web Architecture, Processes, and Organizations or the Modern Enterprise / Martin L. Abbott. – 2st Edition. – Addison-Wesley, 2013.
3. Schlossnagle, T. Scalable Internet Architecture / T. Schlossnagle. – 1st Edition. – Developer's Library, 2010.
4. Mani Krishna, C. Fault-Tolerant Systems / C. Mani Krishna. – 1st Edition. – Elsevier, 2008.
5. The Architecture of Open Source Applications [Electronic resource]. – Mode of access: http://www.aosabook.org/en/. – Date of access: 29.09.2015;

UDC 510

SINGULAR DECOMPOSITION OF MATRIXES IN TASKS

OLGA AGAFONOVA, STEPAN EKHILEVSKIY, NINA GURYEVA
Polotsk State University,Belarus

For any real or complex (m×n) – matrix A with rank r of the matrix A^*A and AA^*, where A^* is obtained from the matrix A by transposition and replacement of components on the complex conjugate is, are symmetric or Hermit with rank r and dimension according to n and m. Note that they are non-negative. Therefore characteristic numbers of such matrixes are the real non-negative numbers.

We designate characteristic numbers of a matrix A^*A through $\rho_1^2, \rho_2^2, ..., \rho_n^2$, considering that $\rho_1^2 \geq \rho_2^2 \geq ... \geq \rho_n^2$ ($\rho_i \neq 0$ when *i=1, 2,…, r*).

It is known that the operator with a symmetric or Hermit matrix A^*A has orthonormal system of eigenvectors $e_1, e_2, \dots, e_n$ respectively on $\rho_1^2, \rho_2^2, ..., \rho_n^2$, then there is such vectors that $A^*Ae_i = \rho_i^2 e_i,\ i = 1, 2, \dots, n$, where

$$\left(e_i, e_j\right) = \begin{cases} 1, \text{ when } i = j, \\ 0, \text{ when } i \neq j \end{cases}.$$