# BENCHMARK-BASED OPTIMIZATION OF COMPUTATIONAL CAPACITY DISTRIBUTION IN A CLIENT-SERVER WEB APPLICATION

Zsigmond MÁRIÁS*, Ádám TARCSI,**, Tibor NIKOVITS***, Zoltán HALASSY****
*Faculty of Informatics, Eötvös Loránd University (ELTE), H-1117 Budapest, Pázmány Péter sétány 1/C, Hungary, tel.: +3630 389 5535, e-mail: zmarias@inf.elte.hu
**Faculty of Informatics, Eötvös Loránd University (ELTE), H-1117 Budapest, Pázmány Péter sétány 1/C, Hungary, tel.: +361 372 2500/1816, e-mail: ade@inf.elte.hu
***Faculty of Informatics, Eötvös Loránd University (ELTE), H-1117 Budapest, Pázmány Péter sétány 1/C, Hungary, tel. +361 372 2500/8475, e-mail: nikovits@inf.elte.hu
****LogiNet Systems Kft., H-1221 Budapest, Vihar u. 5/D, tel. +3620 415 3638, e-mail: zhalassy@loginet.hu

**ABSTRACT**

*There is a common task in web applications when the content of a database table or the result of a query has to be listed on the client side. In traditional implementation the requested amount of data needed for the page is queried and created on the server and is sent to the client. Then, at every interaction a new http request is sent to the server and the database layer generates a new data set to be displayed.*

*Modern web applications can rely on the higher computing power of the client computers. Therefore state-of-the-art applications can transfer more data to the client where data are handled in an array and frontend Javascript executes filtering, sorting, and paging. This approach saves server capacities and omits the network communication lag; at the same time the user interface becomes faster and more responsive.*

*A backend-frontend solution would be useful that distributes the computational tasks according to the available resources on the client side. As the data processing capability is not universal to all clients, benchmark logic has to be applied to adjust client specific data limit.*

***Keywords:*** *web technologies, web frontend, optimization, benchmark*

## 1. INTRODUCTION

There is a common task in web applications when the content of a database table or the result of a query has to be listed on the client side. As the end users of the software are normal human beings who usually cannot use large amounts of information at once, the data are shown in pieces on separate pages. Paging functions are required to iterate between pages, and as usually users are only interested in a subset of data in a specific order, filtering and sorting functions are provided for the users.

This is a frequently used function playing an important role on the user interface and it appears in nearly all web applications.

### 1.1. Several examples

An enterprise system often shows data in the form of lists for the users. Data can be of any kind: invoices, orders, etc. Users access the data on pages; data can be ordered or filtered.

A forum or blog software shows a list of comments on the user interface on pages. User can browse comments and sort them by relevance or date.

E-commerce software lists the products for the users. Users can define different filter conditions and sort products by price to find the products that exactly suits them.

An essential function of online banking systems is account history, where users can browse their activities. The activities are listed on pages and the list can be filtered to credit card payments.

As shown, this is a common task which is used in nearly all web applications.

### 1.2. Requirements

In this section the requirements and tasks are specified for the solution with database query examples. The main objective is to provide a web browsing frontend for a dataset which contains records of the same type. The following features are required:

The frontend shows a certain amount of data at once for the user that can be understood. This means that the dataset is divided into pages, and always one page is shown for the user. If the number of records to show on a page is $k$, then the $N$-th page of the relation R could be queried in the following way:

SELECT * FROM R LIMIT $k$ OFFSET $(N-1)*k$

The frontend provides a function to iterate through the list of the pages. This means that when the user browses a certain page, the previous, next, first and last pages are easily accessible. If the number of records to show on a page is k, and the N-th page of the relation R is shown, then the query for the next action would be:

SELECT * FROM R LIMIT $k$ OFFSET $N*k$

The next feature is filtering the dataset by a certain constraint condition. To keep the notation simple, only the $\alpha = A == a$ condition type is considered in this paper.

However, as the results shown are general, several other condition types could be introduced such as regular expressions, interval constraints, etc. If the table R is filtered with

condition $X == y$, the query could be

SELECT * FROM R WHERE $R.X = y$

The data can be sorted by a certain field and after this action, the first page of the result is shown to the user. In this paper ordering by multiple fields is not considered, but adding the support of this function does not change the basis of our solutions. If a table R is sorted by the column X, the query would be:

SELECT * FROM R ORDER BY X

### 1.3. Regular solutions

#### 1.3.1. On the server side

The above mentioned task has been acute since the first web application. In the beginning the web browser running on the client side was not able to do anything but display the received html document. It was not used to run any business logic control. There were two reasons: the resources of the client were limited and there was no applicable software technology.

This approach executes a listing as the server always sends one page content of the html document. Any interaction on the client side generates and sends HTTP GET parameters to the server and so it requests the next page. Below is an example URL of http request which contains a filter and order command and one page to be displayed.

$[...]?filterColumnId = 22\&filterValue = foo$
$\&orderColumnId = 12\&page = 3\&show = 20$

In turn it becomes a database query which contains the above conditions.

SELECT * FROM R WHERE col_22=foo ORDER BY col_12 LIMIT 20 OFFSET 40

Obviously in this model the server executes all the computation in the database layer and in addition at each interaction the page is reloaded which causes network overhead and it spoils the user's experience.

The up-to-date solution in this context is the same but the increase of client's resources and the AJAX[1] technology allow doing it without reload of the whole page. The filter, order and page conditions are transferred in the http request to the server as earlier but the answer contains only the live data in JSON or XML format. From computation distribution's point of view the solution is the same as it was earlier.

Two important aspects are improved: the network overhead is decreased and the user's experience is also better.

#### 1.3.2. On the client side

The ever growing computation power of client computers and the evolution of Javascript[2] interpreters allow the other approach. The vast majority of the computation is made on the client's side. The server transfers the whole data in XML or JSON format to the front-end. The browser stores data in its memory and a logic written in Javascript will do computation related to data view.

The ultimate advantage of this approach is the relief of resources on the server side. The computation load is taken over to the client side as the database query is implemented once again in the front-end layer. The disadvantage is in the substantially smaller power of the client computer and the program running in the front-end layer is much less optimized as that of the server side. Big data set of hundreds of thousands of records slow down the client in a sensible extent. So there are well defined cases when the approach cannot be applied.

## 2. SUGGESTED SOLUTIONS

In this chapter several solutions of the described effect will be shown. The common point in the solutions is that all of them use a combination of the two basic solutions.

### 2.1. Global threshold

As we saw in the previous chapter, the main problem is that over a certain amount of data the client computers cannot store and handle it in the memory. Therefore in the applications when the developer knows that the amount of data is relatively small, and in almost every case the clients will be able to handle it, the second method can be used, in other occasions the first one. This technique uses one threshold, which can be hardcoded as described, but also can be implemented as a system parameter that can be adjusted.

### 2.2. Benchmark based client-level threshold

For the second approach we first have to notice that the client computers are different and some of them have as strong computational capacity as the server does, while others are really weak. The previous solution ignores this fact and sets up a really low threshold which suits all client computers. Hence, a lot of computation that could be performed by clients will still be performed by the server. To use more of the available capacities a specific threshold has to be set for each client, which is the limit of the data that a particular client can handle. For this purpose a kind of benchmark has to be used to identify this particular value. The whole solution would work as follows:

When the client makes the first access to the system a Javascript benchmark is performed on the client and the result is stored in a cookie value. From this point whenever the client starts to browse a data table, depending on the number of rows in the target database table the first or second technique will be performed. This also means that before every action, the server has to perform a counting action -

SELECT COUNT(*) FROM R WHERE $\{conditions\}$

- and depending on the result the limited data or all data are given to the browser.

### 2.3. Client-based, adaptive threshold

The solution above is fine enough and does the work much better than the classical solutions, but has a drawback. Regarding one client and a database table, the client is going to save more and more computational capacity until a certain point. This certain point is when the amount of data exceeds the maximum number set for that client. Then, the usual approach will work all the time, not saving any resources from the server. At the first glance this seems to be a characteristic of the solution that cannot be avoided. But let us assume:

1. The target database table has [A,B,C,D] columns and contains $N$ rows. The threshold for the particular client is $T = 0,5 * N$.

2. The user queries a whole list, with a filter condition "$A == 1$", the response contains $N/3$ rows, and the first page is queried from the database.

3. The user sets up a sort condition on the previous dataset on $B$ and starts to browse the result, performing 5 paging actions.

4. Then the user makes one more filter condition, "$C == foo$" for example.

In this example the previous solution would perform every action by separate queries from the database. But when the first query ($A == 1$) was performed, the result dataset was already small enough to be handled by the client. Hence, the queries initiated in steps 3 and 4 could be avoided on the server side. So in this solution if the browsed database table contains more rows than the client's threshold value, but the result dataset of a user query to be browsed is below it, all data are sent to the client and the client performs all the queries. As only the queries with filter constraints can decrease the amount of rows to show, we can assume that the query contains a logical formula

$$\alpha = \alpha_1 \wedge \alpha_2 \wedge \alpha_3 \wedge ... \wedge \alpha_n \ ,$$

where $\alpha_i = A_i == a_i$ .

If the result of such a condition is calculated and given to the client all further queries that do not broaden the dataset, are performed on client side.

Paging actions never change the dataset; just show a particular part of it. Sort actions do not change the dataset either, they reorder and show it. Hence, all the calculations related to these actions can be performed on the client side on the dataset stored in the client's memory.

Filter actions do not broaden the dataset as far as the user only defines further filter constraints in form of $\alpha' = \alpha \wedge \alpha_{n+1}$ but does not remove any $\alpha_i$ from $\alpha$. If the user does remove conditions from a query, the calculations have to be performed on the server side, because the domain dataset is not available on the client side.

## 3. ELEMENTS OF THE SOLUTION

### 3.1. Javascript benchmark systems

A very important part of the proposed solution is the proper Javascript benchmarking to identify client capacities. The task is not only to provide a good threshold, but also to do that relatively fast. During the browser war in 2000's[3] one of the key points of the competition was to build the fastest Javascript engine in the browsers. In this competition Javascript benchmark libraries play a really important role.

These libraries check the implementation of different Javascript services and also contain speed check. The normal use of these libraries is to install different browsers on a computer, run them and then compare the services available and the results. For the purpose of the described solution the most important client factors are the amount of memory available and the speed of calculations with arrays.

In the browser war mentioned above, almost every browser developer company created a benchmark suite in which their browser ranks highest. Some of these solutions were examined to choose the one that suits best to the solution.

- Kraken[4] is a Javascript test suite from Mozilla which focuses on realistic workloads and forward-looking applications, such as beat detection scripts, which uses experimental audio APIs, and image processing tools application of Gaussian blur or desaturation of a JPG image using Javascript. A computer with quad-core Core-i7 processor and 8GB ram, running Chrome version 29.0.1547.62m passes the test in 1600ms approximately. Kraken source code can be checked out and altered to run only the important benchmark test cases, which are the JSON and Stanford test cases.

- Google V8 Benchmark is the Javascript test suite by Google, used to optimize Google Chrome web browser. It is superseded by Google's Octane[5] benchmark which replaces the V8 benchmark. According to Google, "Octane v.1 consists of 13 tests, 5 new ones and 8 from the original V8 Benchmark Suite". Octane source code can be checked out and altered to run only the important benchmark test cases which are the Regexp, CodeLoad and Box2DWeb test cases.

- Other benchmark suites that have been examined were Dromaeo and Sunspider which are also Mozilla benchmark suites. They perform several tests, and they run for several minutes on a modern computer.

There were two main points in the selection of the benchmark solution.

The first point is to measure the performance of proper use cases. Some benchmarks contain benchmarks of experimental new built-in libraries that have no relation to our purposes; therefore they have to be avoided.

The second point is that benchmark tests cannot run and put long and high load on the client. Some seconds runtime is acceptable but above that the user experiences it.

However the benchmark has to be run only once, and might be repeated occasionally. Hence, if the runtime is not extremely long, it can be seamlessly integrated in the solution and run in the background.

In the solution Google Octane is used, because the code can be downloaded and altered to remove the useless test cases which benchmark Google Chrome specific or irrelevant features. This way the results arise quickly and they are general.

### 3.2. Software components

The whole solution consists of the following logical components:

- The *backend*'s most important function is to handle the database, including the implementations of database queries that produce datasets according to the client actions. It produces JSON objects or XML documents that are sent back to the frontend. It also produces metadata for the frontend to generate filter conditions. It consists of database specific queries and functions written in the actual language of the backend system - Java, PHP, C#, etc.
- *Frontend view* is the software module that generates the tables in html format from the dataset to be shown. It is written in Javascript, standard solutions such as jQuery[6] Datatables plugin are an obvious choice.
- The *benchmark module* is the Javascript library introduced in the previous chapter that discovers the client's computational capacities. This module also ensures that the results are stored in cookies and updated after a certain period of time if necessary.
- The *frontend computations* layer is the "backend of the frontend", which plays three important roles. It re-implements all calculations in Javascript that are available in the database layer. This implementation is made in Javascript and it mainly contains operations with Javascript arrays. It decides if a certain action has to be calculated by the server or the result can be obtained from the currently available data stored in the memory. In this feature the previous and current user actions as expressions are compared and the currently available dataset is analysed. Depending on the result of the decision the calculations are performed by the according software component.

## 4. SIMULATIONS

A simulation software suite was created to analyse the possible advantages of the suggested solution. The basic idea was that when a single server is requested parallel by multiple clients and the number of requests grows, after a certain point the server will not be able to fulfil the requests in acceptable time. In this situation the end user experiences that the service is out of order.

On the other hand if the dataset given to the client exceeds its capacities, then the user has the same experience, although the server is working. The expectation is if the solution described above is used, then more clients can be served in acceptable time without giving too much data to any of them.

### 4.1. Test parameters

The benchmark is a simulation of parallel requests from a single server with multiple clients. The simulation parameters were:

- Number of clients.
- The minimum and the maximum available memory of client dedicated to the request.
- The average time a user browses the result before makes a new request.
- The average skipped requests due to the client caches the database contents and do not need to query the server for it.
- The CPU time the server requires to compute an answer.
- The time the client requires to compute an answer from cached database values.
- The average requests a client makes during a session.
- The number of requests the server can serve parallelly. Extra clients are waiting in the queue in a first-come-first-served basis, till a processing slot opens up because a previous client got served.

The benchmark randomizes the start time of requests between 0 and the average request interval. There is a 5% error of the CPU time required for a process to be completed. There is a 5% fuzz on the next request time from the previous one. The memory size of the clients rolled randomly, with a uniform distribution.

### 4.2. Test assumptions

The benchmarks worked with the following assumptions:

- The data sent to the client are smaller than a user would notice with the client's network bandwidth.
- The clients are equally strong.
- The server serves the clients with equal CPU power parallelly, does not work on anything else, and 100% of the power is used when there is work to do.
- If the client has enough memory to fit the database in it, the server will use 0 CPU power to give the data to the client.

### 4.3. Test results

On the chart shown in figure 1 three different use cases can be seen. In the first use case, all clients were set to have zero memory, which results that all calculations are performed on the server side. On the second use case clients are generated with random capacities with standard distribution - some of them are almost infinitely powerful, some of them have no memory and the majority of them fall in between these values. On the third use case clients are generated the same way but all data are always sent to the clients. If the data exceed the limit of a client, then it is considered to be an unacceptable run.

In the figure it is visible that the number of unacceptable runs is the highest in the server only solution. The client only solution has significantly fewer of them, but the solution proposed in this paper outperforms both techniques.

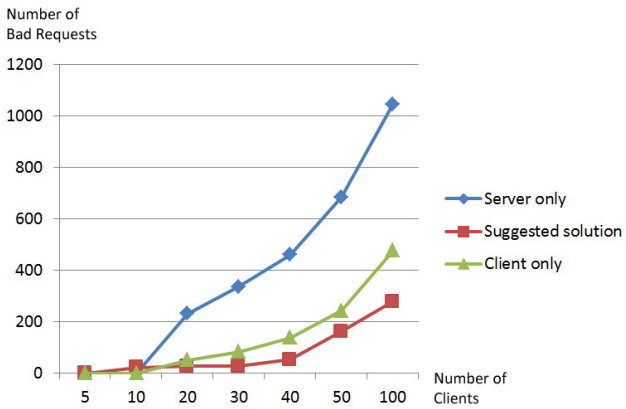


**Fig. 1** Benchmark results

## 5. CONCLUSION

As shown in section 4 the proposed solution can significantly increase the number of parallel requests that can be handled by a server with certain computational capacity. However, there are some consequences for the solutions that have to be considered in the future.

The main problem is that for clients with high computational capacities, a lot more data are sent through the network than actually used. When the network bandwidth is the bottleneck of a system, this solution should not be used. But when network bandwidth is high and clients use datasets for multiple actions, which behaviour is quite natural in business software, the implementation of the proposed solution can be advantageous.

## 6. FURTHER WORK

In the future, some enhancements are planned for the system. The first planned enhancement is a test, for filter queries with multiple conditions.

If a dataset is too big for a client and a filter condition $A == a \quad \wedge \quad B == b$ is set by the user, and $A == a$ or $B == b$ conditions already result a dataset that can be handled by the client, then one of the filter conditions can be eliminated on the server side. The $A == a$ condition could be calculated by the server, while $B == b$ on the client side. Actually, this enhancement does not count a lot but if we consider that fact that in this case a bigger dataset is in the client's memory, and $B == b$ condition is removed or changed to $B == b_1$, then the client will be able to calculate that query.

The second enhancement is a promising idea about including a subsystem that analyses the user behaviour and stores which filter conditions are frequently used[7]. In the previous example the suggestion was to calculate $A == a$ and let the server calculate $B == b$ condition. It is easy to see that if $B == b$ is changed less frequently than $A == a$, then the profit would be higher if $B == b$ is calculated on the server side. If a database contains the frequently used filter conditions, then a decision automatism can be included.

User behaviour analysis can be also used to identify which tables are often used for multiple user actions and which are used for one query at each access. This information can also play a role when the system makes a decision between sending a lot of data to the client or keeping the computations on the server side.

## REFERENCES

[1] GARRETT, J. J.: (18 February 2005).“Ajax: A New Approach to Web Applications”. AdaptivePath.com. Retrieved 19 June 2008.

[2] FLANAGAN, D. – FERGUSON, P.: (2006). JavaScript: The Definitive Guide (5th ed.). O'Reilly & Associates. ISBN 0-596-10199-6.

[3] ATWOOD, J.: (2007-12-19). “The Great Browser JavaScript Showdown”. Retrieved 2008-09-06.

[4] “Release the Kraken”. The Mozilla Blog. Retrieved 2013-08-08.

[5] “The Benchmark - Octane - Google Developers”. Retrieved 2013-08-08.

[6] JQuery: finding your way through tangled code - E McCormick, K De Volder - Companion to the 19th annual ACM , 2004 - dl.acm.org

[7] CHEN, C. – CHEN, M. – SUN, Y. (2002)., PVA: A Self-Adaptive Personal View Agent. Journal of Intelligent Information Systems, p.173-194, 2002

Received December 1, 2013, accepted December 21, 2013

## BIOGRAPHY

**Zsigmond Máriás** was born on March 13th, 1983. He graduated (MSc) in 2007 as Program Designer at the department of Informatics at Eötvös Loránd University, Budapest. He is the CIO of a IT company and part-time lecturer at Eötvös Loránd University. His main fields of in-

terest are information systems design, petri nets, workflow modeling, data model design and web applications.

**Ádám Tarcsi** received an MSc teaching degree of Informatics at Eötvös Loránd University Faculty of Informatics in 2004. His PhD researches are based in Web Information Systems. He has experiences in SAP, Web development and in business analytics. In 2008 he becomes an Assistant Professor at the Department of Media and Educational Informatics. Now he is the Strategic Advisor at ELTE Faculty of Informatics.

**Tibor Nikovits** received an MSc degree in mathematics from Eötvös Loránd University Budapest in 1988 and an MSc degree in economics from Budapest Economical University in 2003. Since 1997 he is working as a lecturer at the Information Systems Department of Eötvös Loránd University Budapest. His main fields of interest are database theory, computer networks, information systems, especially financial and business related information systems.

**Zoltán Halassy** was born on February 2nd, 1983. He graduated in 2011 as Program Designer at the department of IT at Eötvös Loránd University in Budapest. He is a lead developer of a company where part of his job is to optimize program codes which handles large datasets.