Narrowing Down the Debugging Space of Slow Search Response Time

Dapeng Liu[†], Youjian Zhao[†], Dan Pei[†*], Chengbin Quan[†]
Qingqian Tao[‡], Pei Wang[‡], Xiyang Chen[‡], Dai Tan[‡], Xiaowei Jing[§], Mei Feng[§]
[†]Tsinghua University [‡]Baidu [§]PetroChina
[†]Tsinghua National Laboratory for Information Science and Technology (TNList)

***Abstract*—When using search engines, users often care about search response time (SRT) in addition to result accuracy. It is thus the operators' responsibility to closely monitor and improving SRT. The first critical step of improving SRT is to pinpoint the root causes of slow SRT. However, this task is very challenging because SRT can be impacted by many factors, e.g., networks, data centers, browsers, and the page content.**

In this paper, we propose FOCUS, a systematic framework to narrow down the debugging space of slow SRT by identifying the bottleneck of slow SRT regarding various factors. The bottleneck provides operators more specific direction for further investigation. We deployed FOCUS in a global top search engine. Based on the output of FOCUS, operators successfully identified four potential causes which would not have been easy to find without FOCUS. Our what-if simulation analysis shows that, the proposed solutions, focusing on these bottlenecks, can improve SRT significantly, and they are more effective than some ad hoc solutions.

I. Introduction

Search engine is no doubt one of the most prevalent applications of the Internet. Billions of queries are launched from all over the world everyday, and then handled by search engines such as Google, Baidu, Yahoo, Yandex, and Bing [1]. Operating such a giant search system is very challenging, and operators have to work very hard to satisfy dozens of KPIs (key performance indicators). Among them, search response time (SRT) is one of the biggest concerns for search providers [2]. SRT refers to the user perceived waiting time between when a query is submitted and the time when the result page is fully rendered. As such, SRT has a measurable impact on users' experience as well as providers' profit. [3], [4] found that less than half a second increase of SRT can lead to 0.6% fewer searches and 1.2% drop in revenue. As a result, operators are responsible for monitoring and improving SRT, especially the slow SRT, so that it can satisfy the growing requirement. For example, the search engine we studied requires the 80^{th} percentile of SRT less than 1 second.

Recently, many acceleration solutions have been proposed [5], [6], [7], [8], [9], [10], [11] . They aim to fix particular problems in the fourth step. Yet, a key missing part before applying a solution is to identify the bottleneck of slow SRT. In particular, we want to answer the following two questions: Under which conditions queries are slow? Which components of SRT is slow? These two questions can help operators debug slow SRT.

In this paper, we propose a novel systematic framework, called FOCUS, to systematically answer the above two questions. FOCUS intends to automatically identify the *bottleneck conditions*, and the *bottleneck SRT components*. The results, outputted by FOCUS, provide operators specific investigation directions and enable them to further identify the root causes. For example, if we find that many queries triggering ad are responded slowly, and their DOM (document object model) load time is long, operators should investigate whether the modual regarding ad is inefficient, or contains some bugs.

The task of FOCUS in practice is challenging due to the following aspects. First, SRT can be affected by many factors such as servers, networks, browsers, and users' devices. Second, since these factors inherently overlap each other, it is difficult to identify which factor is responsible for the slow SRT. For example, a condition that Chrome runs on a less powerful device. Third, the output should be specific and straightforward for operators. For example, the output of traditional clustering methods like k-means do not have clear boundaries, thus unintuitive.

To tackle these challenges, we first develop a multi-dimensional hierarchy clustering to provide clear and meaningful boundaries of the bottlenecks. This ensures that the clusters we find out are specific for operators. Then we leverage a technique called hierarchical heavy hitter (HHH) [12], that has been commonly used to locate iceberg in network traffic. This technique can help identify in the multi-dimensional hierarchy which clusters are the real bottlenecks and which ones are redundant. Once bottleneck clusters have been found, we design a method based on Occam's razor to further determine which SRT components can best explain the slow SRT.

We deploy FOCUS in one of the global top search engines. Based on the bottlenecks identified by FOCUS, operators successfully locate four causes of slow SRT and propose solutions. Our what-if simulation further demonstrates that, the solutions focusing on the bottlenecks by FOCUS are much more effective than ad hoc solutions in improving SRT. Some of these ad hoc solutions were actually being considered by the operators for deployment in the studied search engine before using FOCUS. These results highlight the value of FOCUS in the field of debugging slow SRT.

The remainder of the paper is organized as follows. Section II provides the basic background of SRT and our problems. Section III describes the details of FOCUS. Section IV shows the results of FOCUS over real data and the simulation. Section V reviews the related work, and Section VI concludes the paper.

II. Background and Problem

A. About SRT and Requirement

To better understand SRT, we first introduce the events happened after submitting a query. Instead of discussing the details, we here only provide a simplified view to build high level intuitions. Fig. 1 shows five steps. (1) When query is submitted (if the result is not cached by the client), the host name of the search engine will be resolved by the the provider's DNS. The DNS responds the IP address of a close data center. (2) Then the browser sends a query to the search data center. The data center conducts a series of complex processes, such as results ranking, ad strategies, and page constructing, before sending the result page to the browser. (3) The browser starts parsing the page and loading DOM. (4) Embedded images of the page are

* Dan Pei is the corresponding author.

978-1-4799-7575-4/14/$31.00

acquired. These images are geographically distributed among plenty of CDN (content delivery network) nodes for accelerating. (5) At last, the page is completely visible for the user.

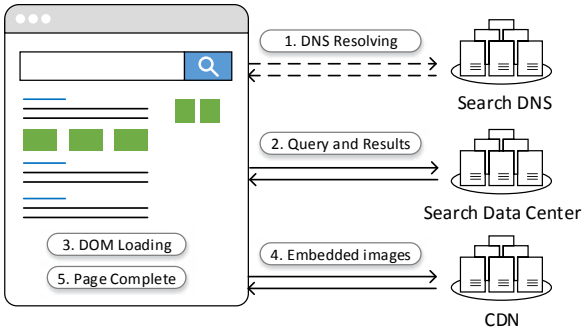

Fig. 1. Typical steps of a search.

The time between submitting a search query and the result page being completely rendered is called search response time (SRT). SRT is the user perceived waiting time. It thus has significant influence on users' experience, and can further affect providers' revenue. Hence, operators need to carefully monitor and improve SRT, so that it can satisfy the SRT requirement. The requirement is usually to focus on percentiles rather than the average, as percentiles can describe how many queries should be served as expected. For example, the 80^{th} percentile of SRT less than $1000ms$ means that the SRT of 80% of queries should be no more than $1000ms$.

B. Data Collection

In this paper, we leverage two categories of data for each query, i.e., the SRT measures and impact factors[1]. As shown in Fig. 2, the SRT measures include SRT and its components, and the impact factors are several conditions that can potentially affect SRT. Their details are described later.

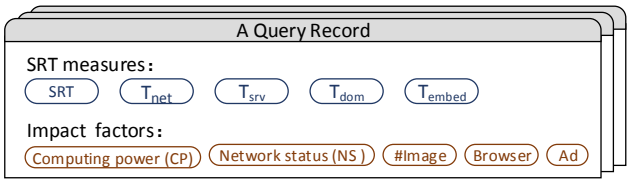

Fig. 2. The content of query records.

Though web servers are usually configured to maintain web access log, this sort of data lacks detail. For example, the response time logged is actually the server processing time, which is only a part of SRT. Additionally, only a few impact factors is logged, e.g., browser types. Therefore, we build our data collection agent to gather necessary information for identifying SRT bottlenecks. The agent takes advantage of a technique called web instrumentation, which is commonly used by search providers [2]. Specifically, the agent is a piece of javascript codes, that equipped into the pages and run at user side. During a search, it is responsible for recording the SRT and its components, as well as several impact factors. It can also receive information from servers, such as the server processing time. Finally, the query record is logged by the agent and sent back to storage servers. The advantage of the agent is that it can be deployed at large scale. But as it can introduce extra overhead for users, we have to simplify the agent and give up fine grained data. Now, we characterize how the query records are measured by the agent.

[1]In compliance with confidentiality constraints, the specific names and values in this paper are anonymized and normalized.

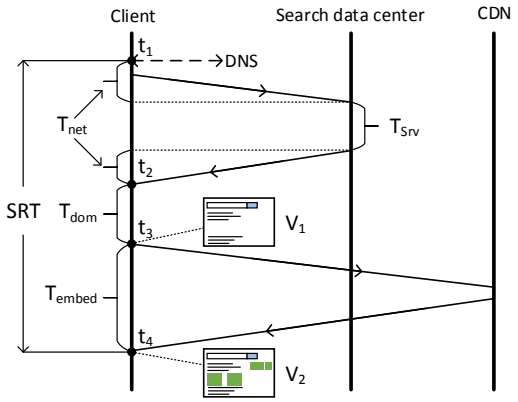

Fig. 3. The timeline of a search. During the search, four time points are recorded. SRT and four components are calculated based on them.

1) SRT and Components: Fig. 3 shows the timeline of a search. Four critical time points are recorded by the agent. When a user launches a search query, we record t_1. After an alternative DNS resolving, the query is sent to a search data center and the result page is returned. When the page is completely received, we record t_2. Then the browser starts parsing and loading DOM, and after the DOM has been fully loaded, we record t_3. At this time, the page appears like V_1. The main frame and the result list are visible, but the images are empty. Next, the embedded images are acquired from CDN and rendered into the pages. When this process is finished, the page is completely visible like V_2, and we record t_4 then.

Though we omit more details, these four deliberate time points are sufficient to calculate SRT and its key components. The SRT for a query is time between t_1 and t_4. The server processing time T_{srv} is recorded by the server and sent back to the agent. Then we have $T_{net} = t_2 - t_1 - T_{srv}$, including the DNS time and the network transmission time of the query and the result page. The DOM load time of the browser is $T_{dom} = t_3 - t_2$ and the time consumed by other embedded elements is $T_{embeded} = t_4 - t_3$. We see that SRT is the sum of T_{net}, T_{srv}, T_{dom}, and $T_{embeded}$. These four major components reflect different parts of SRT, and are useful for debugging slow SRT.

2) Impact factors: Another job of the agent is to gather impact factors, that can potentially affect SRT. The factors we collect are shown in Fig. 2. (1) Computing power (abbreviated CP): this factor is measured by a piece of benchmark code. The code is executed by the users' browser, and its runtime is used to estimate CP. So, CP captures the synthetical performance of CPU, browser, etc. (2) Network status (abbreviated NS): it intends to measure the network performance of users. We approximate this factor by T_{net}. As the DNS server takes advantage of efficient cache, the DNS revolving takes very short time at the DNS server. Besides, the page size does not vary a lot, so we ignore the error caused by different sized pages. (3) $browser$: we distinguish browsers by their basic types and major version numbers, and we have 8 major browsers, denoted as $x1, x2, ..., x8$. For nondisclosure reasons, we cannot report their specific names. (4) $\#image$: as many images can considerably slow down the page load, we record the number of embedded images of each page. (5) ad: the page with ad contains more complex codes to realize business function and enable advanced interaction effects. We simply use "yes" or "no" to represent whether a page includes ads or not.

We collect the above representative factors based on domain knowledge. But our analysis framework is not limited to the factors we used, and can be easily extended to other factors.

C. Problem Statement and Goal

First, we define the gap to the SRT requirement. Let $R(p\%, t)$ be the SRT requirement, which means reducing the p^{th} percentiles of SRT to t milliseconds. We call t the desired SRT. $R(p\%, t)$ can also be interpreted as that the SRT of at least $p\%$ of queries should be no more than t. The requirement can be reflected by a CDF. As shown in Fig. 4, the dashed line indicates a SRT distribution that satisfies the requirement. We call a query *slow query* if its SRT is larger than t, or *fast query* otherwise. Then $R(p\%, t)$ is equivalent to owning $p\%$ fast queries. Yet within the current query records, we only have $p'\%$ fast queries, as shown by the solid line in Fig. 4. Consequently, to achieve the requirement, at least $\Delta = p\% - p\%'$ slow queries need to be changed to fast ones, after considering the case of fast queries changed to slow ones. We use Δ to denote the gap between current SRT and the requirement.

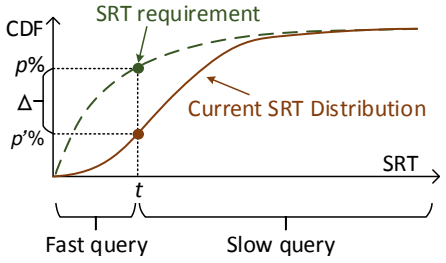

Fig. 4. SRT requirement.

Subsequently, operators need to debug at least Δ slow queries, in other words, locating their causes. Then they can take action to accelerate them[2] and filling in the gap, e.g., deploying accelerating solutions or fixing bugs.

Our goal is to help narrow down the debugging space. Specifically, we first aim at identifying what kinds of impact factors generate at least Δ slow queries. These factors are namely *bottleneck clusters*. Then, in each bottleneck clusters, we further find out which components can best explain the slow SRT. These components are namely *bottleneck components*. These bottlenecks can provide operators a more specific debugging direction to identify the root causes. In this paper, we use the requirement of the search engine we studied: $R(80\%, 1000ms)$. Here, $1000ms$ is the time that can potentially interrupt users' flow of thought [13] and the time that search engines like Google [14] and the one we studied try to avoid.

III. Design

At a high-level, as shown in Fig. 5, FOCUS take as input the SRT requirement and the query records, and identify the bottlenecks of SRT as output. In the first steps, we hunt for the clusters of impact factors that contains at least Δ slow queries. To ensure intuitive clusters, we employ a novel schema to define the boundaries of clusters on each factors. In nature, all these clusters are organized in multi-dimensional hierarchy, and we exploit HHH to identify the bottleneck clusters in this structure. Afterwards, for each bottleneck cluster, we determine bottleneck components based on Occam's razor. The basic idea is that the most succinct one should be the best explanation. The above bottlenecks are the output of FOCUS. Operators can further investigate along the directions of the bottlenecks to locate potential causes, and design solutions. We also conduct what-if simulation to evaluate the effectiveness of these solutions (Section IV).

[2]In this paper, accelerating slow queries refers particularly to improve them to fast ones, in other words, reducing their SRT from $> t$ to $\leq t$.

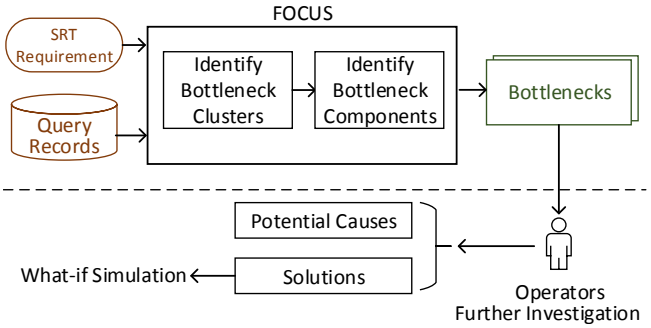

Fig. 5. High level overview of FOCUS.

A. Defining Cluster

The impact factors are measured with either classes such as $browser$ or numeric values such as $\#image$ and CP. Generally evaluating all possible subset of each factor will generate unnecessarily large number of clusters. Moreover, such clusters would be too spotty for operators to understand. Instead, we first group the values of each factor into meaningful units, then define clusters upon these units. This kind of clusters are much more intuitive to be used.

For ad, it has just two units, $ad = yes$ or $ad = no$, as it appears originally. For browser, we treat some browsers as the same unit, as they have the same compatibility and similar performance. As a result, there are five units out of 8 browsers, $U1 = \{x1, x2\}$, $U2 = \{x3\}$, $U3 = \{x4\}$, $U4 = \{x5, x6, x7\}$, and $U5 = \{x8\}$.

On the other hand, for the other three numeric valued factors, i.e., $\#image$, CP, and NS, we divide them all into four units, which are fine-grained enough for our problem, and they also have clear meanings, such as the $\#image$ can be $fewer$, few, $many$, and $more$. Now we take $\#image$ as an example to show our determination of each unit. For practice, we design an automatic splitting schema rather than divide units statically. The schema starts with the observation that when we decide the $\#image$ is many or few, we actually consider SRT. For example, if the SRT of queries wit 10 images is faster than the SRT of overall queries, then 10 images seem not a lot; on the contrary, if 15 images can result in a slower SRT than the overall SRT, 15 images are many. As such, we first decide SRT ranges in current queries, which is quite straightforward, and use these ranges to determine the units of each factors.

Here, we also use p^{th} percentile as the metric of SRT. Let t' be the p^{th} percentile of SRT in current queries. Then we have four SRT ranges $(0, t' - \epsilon]$, $(t' - \epsilon, t']$, $(t', t' + \epsilon]$, and $(t' + \epsilon, \infty)$, representing varying deviations from t'. Here ϵ is a time interval and can be set according to demands, e.g., $100ms$ or $10\% \cdot t'$. Note that we do not use the desired SRT t to divide SRT, because t is relatively small and independent of current SRT, thus the SRT ranges based on t cannot decide the units appropriately.

Subsequently, for an arbitrary $\#image$, it is called $fewer$, few, $many$, or $more$ respectively if the p^{th} percentile of its SRT is within $(0, t' - \epsilon]$, $(t' - \epsilon, t']$, $(t', t' + \epsilon]$, or $(t' + \epsilon, \infty)$. By this means, the four units of $\#image$ are dynamically determined and also meaningful.

Another practical issue is that from Fig.6 (a), we see that though more images can inflate SRT more significantly, the SRT is not monotonically increasing with $\#image$ from the perspective of real data. This fluctuation can be due to other confounding factors. As a result, strictly following the above dividing schema can make the $\#image$ in the same unit inconsecutive. For example, in the unit of $(0, t' - \epsilon]$, $\#image$ can be 0, 1, 2, and 6, as the SRT of 6 images is a small valley and fall into $(0, t' - \epsilon]$ too. This kind of units is actually confusing for operators. To avoid this, we employ a trick based on

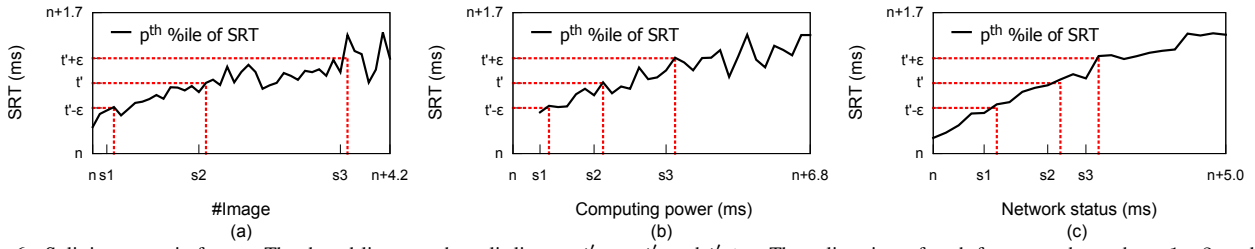

Fig. 6. Splitting numeric factors. The dotted lines are the split lines at $t' - \epsilon$, t', and $t' + \epsilon$. The split points of each factor are denoted as $s1$, $s2$, and $s3$.

the assumption of that $\#image$ and SRT are positively correlated. Specifically, instead of checking the SRT of each $\#image$, we decide three split points of $\#image$. As shown in Fig.6 (a), along SRT axis, we draw three split lines at $t' - \epsilon$, t', $t' + \epsilon$. The $\#image$ most close to the first cross point of each split line from left side is defined as the split point, which are $s1$, $s2$, and $s3$. These three split points finally generate four consecutive units of $\#image$, that is $[0, s1]$, $(s1, s2]$, $(s2, s3]$, $(s3, \infty)$. CP and NS are handled in the same way, as shown in Fig.6 (b) and (c), except we preliminarily discretize their values into small equal-sized bins, e.g., bins of 20ms. The four units for CP and NS are both called $better$, $good$, bad, and $worse$ respectively. Notice that units can be empty as well. For example, the first unit of CP has only one bin, and a larger ϵ will make it empty.

Eventually, a cluster is defined on each factor by either a particular unit or wildcard * for all possible units. For example, cluster $C = (\#image = many, browser = U1, CP = *, NS = *, ad = *)$ represents the queries originated from browsers of $U1$ and their numbers of embedded images are $many$. For brevity, we omit those factors with *, so the cluster C can be simply denoted as $(\#image = many, browser = U1)$. We also use $(*)$ to denote the cluster with $*$ on every factors.

B. Bottleneck Cluster and Identification

All the clusters are organized into a 5-dimensional hierarchy (since we have five factors). Now, we introduce some basic concepts of multi-dimensional clusters. Fig. 7 shows an example of 2-dimensional clusters. The two dimensions are a and b. In this structure, if a cluster is more specific than another cluster, they are linked by one or more directed edge(s), from the general one to the specific one, e.g., $a1 \rightarrow a1b1$. For cluster C, we name the cluster directly linked from C the *child* of C, and name the set of clusters that can be reached from C via one or more edge(s) the *descendants* of C. Correspondingly, C is called the *parent* of its children and the *ancestor* of its descendants. Within this multi-dimensional hierarchical structure, a cluster can have multiple children and parents along different dimensions, e.g., for $(*)$, it has children $a1$ and $a2$ along dimension a, and children $b1$ and $b2$ along dimension b; for $a1b1$, it has two parents $a1$ and $b1$ from dimension a and b respectively.

Let $\mathbf{S}_C$ be the number of slow queries in C (relative number w.r.t. the total number of queries). Basically, a bottleneck cluster refers to the cluster with no less than Δ slow queries, i.e., $\mathbf{S}_C \geq \Delta$. However, the nature of multi-dimension of clusters can lead to a lot of redundant bottleneck clusters under this definition. This is because a cluster is the superset of its descendants, and its slow queries must be no less than the ones contained by its descendants. For example, $\mathbf{S}_{b1} = 10\% \geq \mathbf{S}_{a2b1} = 6\%$. As such, according to the basic definition, many clusters in Fig. 7 ($\Delta = 5\%$) are bottleneck clusters as they contain more than 5% slow queries. Yet, reporting all these clusters are more than necessary, and some of them are too general to narrow down the debugging direction, e.g., reporting $(*)$ is helpless at all.

To overcome this drawback, we introduce the concepts of *duplicate* and *valid* number of slow queries to help define bottleneck clusters in multi-dimensional hierarchy. This definition is inspired by HHH [12]. The main idea is to attribute slow queries to descendant bottleneck clusters rather than ancestors, as descendant clusters are more specific.

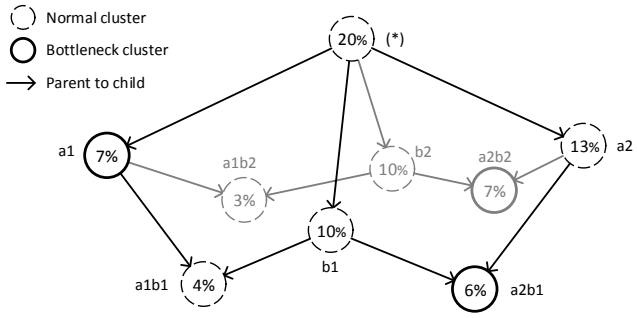

Fig. 7. An example of 2-dimensional clusters. The two dimensions are a and b respectively. We use abbreviation to denote clusters, e.g., $a1b1$ is short for $(a = 1, b = 1)$ and $a1$ is short for $(a = 1, b = *)$. Each circle represents a cluster. The number in the circle is the (relative) number of the slow queries contained by the cluster. In this example, we set $\Delta = 5\%$.

The duplicate number of slow queries of a cluster C is denoted by $\mathbf{D}_C$, and it represents the number of slow queries that have already been attributed to the descendants of C and C . Thus, these slow queries should not be counted repeatedly by the ancestors of C. We also define $\mathbf{D}_C^d = \sum \mathbf{D}_{C'}$, where C' is the children of C along dimension d. If C has no children along dimension d, we treat $\mathbf{D}_C^d$ as 0. Then the valid number of slow queries of C is defined as $\mathbf{S}_C^\star = \mathbf{S}_C - \max\{\mathbf{D}_C^d | d \in D\}$, where D is the dimension set, i.e., five factors in our problem. Finally, we have the following definition:

$$C = \begin{cases} bottleneck\ cluster & , if\ \mathbf{S}_C^\star \geq \Delta \\ normal\ cluster & , if\ otherwise \end{cases}$$

We can see that a bottleneck cluster should still contain enough slow queries after excluding those attributed to descendant clusters. For bottleneck cluster C, we attribute its slow queries to it by setting $\mathbf{D}_C = \mathbf{S}_C$, so that its ancestors will not further recount them; for normal cluster C, as no more slow queries are attributed, we set $\mathbf{D}_C = \max\{\mathbf{D}_C^d | d \in D\}$ to pass on only the slow queries that have been already attributed.

In this definition, the slow queries of a normal cluster can be counted by its multiple ancestors of different dimensions, yet this should not be regard as redundancy. Because the bottleneck clusters of different dimensions actually uncover different problems.

Based on the above bottleneck cluster definition, one can see that the bottleneck clusters are identified bottom up in nature. We use the example in Fig. 7 to show how it works. First, we evaluate the leaf clusters, $a1b1$, $a1b2$, $a2b1$, and $a2b2$. As they do not have any children, we have $\mathbf{S}_i^\star = \mathbf{S}_i$ for them. For these four clusters, only

$\mathbf{S}^{\star}_{a2b1} = 6\% \geq \Delta$ and $\mathbf{S}^{\star}_{a2b2} = 7\% \geq \Delta$, so $a2b1$ and $a2b2$ are identified as bottleneck clusters. Accordingly, we set $\mathbf{D}_{a1b1} = 0$, $\mathbf{D}_{a1b2} = 0$, $\mathbf{D}_{a2b1} = 6\%$, and $\mathbf{D}_{a2b2} = 7\%$. Then come to the next level of $a1$, $a2$, $b1$, and $b2$. For $a1$, it has children only on one dimension (i.e., dimension b). So we have $\mathbf{D}^{b}_{a1} = \mathbf{D}_{a1b1} + \mathbf{D}_{a1b2} = 0$, and $\mathbf{S}^{\star}_{a1} = \mathbf{S}_{a1} - \mathbf{D}^{b}_{a1} = 7\% > \Delta$. As such, $a1$ is a bottleneck cluster, and set $\mathbf{D}_{a1} = 7\%$. For $b1$, similarly, we have $\mathbf{D}^{a}_{b1} = \mathbf{D}_{a1b1} + \mathbf{D}_{a2b1} = 6\%$, then $\mathbf{S}^{\star}_{b1} = \mathbf{S}_{b1} - \mathbf{D}^{a}_{b1} = 4\% < \Delta$. So $b1$ is a normal cluster, and set $\mathbf{D}_{b1} = 6\%$. Likewise, $a2$ and $b2$ are normal clusters too. The last root cluster $(*)$ has children along two dimensions. We first calculate $\mathbf{D}^{a}_{(*)} = 20\%$, and $\mathbf{D}^{b}_{(*)} = 13\%$. Then we get $\mathbf{S}^{\star}_{(*)} = \mathbf{S}_{(*)} - \max\{\mathbf{D}^{a}_{(*)}, \mathbf{D}^{b}_{(*)}\} = 0 < \Delta$. As a result, $(*)$ is not qualified for a bottleneck cluster.

The bottleneck clusters can be identified by a typical post-order traverse of the multi-dimensional tree, such as [12]. Due to the limitation of space, we omit the particular algorithm.

C. Bottleneck Components

Once a bottleneck cluster has been identified, we attempt to find out bottleneck components. Our definition of bottleneck components is based on Occam's razor, which is also used by [15]. It suggests that the most succinct components that can explain the extra SRT should be the bottleneck components. Now we introduce the basic idea via a mock example.

Table I shows each component of the fast queries and the slow queries in a bottleneck cluster. Suppose that the desired SRT $t = 1000ms$, then the average SRT of the slow queries is $500ms$ longer than that. Ideally, we should identify which components of slow queries can best explain this extra $500ms$. To achieve this, we should first determine the extra time each components of the slow queries take. This step requires the desired time of each component to compare. However, only t rather than its components is provided. In order to tackle this problem, we exploit the fast queries to estimate those components of t. For example, in Table I, for fast queries, T_{net} is $100ms$ and the SRT is $500ms$. Compare with slow queries, we see that while the SRT increases by $1000ms$, T_{net} increases by $400ms$. We assume that each component is linearly increases as SRT, so that we can estimate that the increase of T_{net} accounts for 40% of the total increase of SRT. As a result, for $t = 1000ms$, which is $500ms$ longer than the SRT of fast queries, we can estimate the increase of its T_{net} as $500 * 40\% = 200ms$ and finally we have its $T_{net} = 100 + 200 = 300ms$. Similarly, we can estimate other components of t, and they are shown in Table I.

Based on the estimated components, we identify which components can explain H of this extra $1300ms$. H is a threshold of explanatory power. The rule of thumb is that H should neither be small (e.g., 30%) nor over large (e.g., 90%), which can both lead to less helpful results. In our problem, we set $H = 80\%$ and it works well. The last column in Table I shows the extra time each component of slow queries takes when comparing with that of estimated ones. Then, according to Occam's razor, the most succinct set of components, whose extra time is at least $H \times 1300ms$, is the bottleneck components. In this example, the bottleneck components are T_{net} and T_{embed}. This is because the sum of their extra time is $425ms$, longer than $500 \times 80\% = 400ms$, and they also the most succinct set (with two components) to satisfy the this condition.

Actually, under the linear increasing assumption, explaining t is equivalent to explaining the SRT of fast queries. For example, in Table I, if we try to explain why the SRT of slow queries is $1000ms$ longer than that of fast queries, we would get the same results. Hence, we formally define the bottleneck components by explaining the SRT difference between slow queries and fast queries.

TABLE I
THE AVERAGE TIME (MS) OF THE COMPONENTS OF FAST QUERIES AND SLOW QUERIES IN A BOTTLENECK CLUSTER. COLUMN 4 IS THE ESTIMATED COMPONENTS OF THE DESIRED SRT. THE LAST COLUMN IS THE DIFFERENCES BETWEEN THE COMPONENTS OF SLOW QUERIES AND ESTIMATED ONES.

Components	Fast queries	Slow queries	Estimation	Extra time (ms)
T_{net}	100	500	300	200
T_{srv}	250	300	275	25
T_{dom}	100	200	150	50
T_{embed}	50	500	275	225
SRT	500	1500	1000	500

Given a bottleneck cluster, let δ be the difference between the average SRT of the slow queries and that of the fast queries. Similarly, for a component T_i, where $i \in \{net, srv, dom, embed\}$, let $\delta(T_i)$ be the difference between T_i of the slow queries and T_i of the fast queries. The explanatory power of T_i is denoted as $EP(T_i)$, and $EP(T_i) = \delta(T_i)/\delta$. The bottleneck components is a set of components, denoted as BC, that satisfies $\sum_{T \in BC} EP(T) \geq H$, at the same time, minimizing $|BC|$ (i.e., succinctness). While there exist multiple qualified BC, we further select the BC with the maximum of $\sum_{T \in BC} EP(T)$.

Since there are only four components, we identify bottleneck components using a brute force approach of enumerating all possible candidate sets. One can design more efficient algorithm, such as the greedy algorithm proposed in [15], to deal with large-scale data.

IV. EVALUATION

We have deployed FOCUS in a global top search engine. During one week in October, 1% of the provider's queries are collected by our agent (as described in Section II-B). Overall, there are over 28 million query records.

We first present the bottlenecks identified by FOCUS from those query records, and discuss our observations. Then we show the further case studies on the results conducted by the operators of the search engine. It turns out that those bottlenecks successfully help the operators pinpoint some potential causes, and come up with solutions to fix them. Last, we perform what-if simulation of these solutions. In comparison, several ad hoc solutions are also simulated, some of which were actually proposed by the operators before FOCUS. The simulation results prove that the solutions focusing on the bottlenecks can improve SRT strikingly; conversely, the improvement of those ad hoc solutions are much less.

A. Results and Observations

FOCUS identify bottlenecks based on the real SRT requirement of the search engine. The requirement focus on the 80^{th} percentile of SRT. The desired SRT is not shown due to confidentiality. Under this requirement, 75.4% of the queries are fast queries and the other 24.6% are slow queries. As such, the gap to the requirement is $\Delta = 4.6\%$ slow queries.

The bottlenecks identified are listed in Table II, ranked by the fraction of slow queries in each bottleneck (the column of %Slow query). Each row displays one bottleneck, including the clusters, components, and other details. We now characterize some key observations from the results.

Observation 1: Focusing on the bottleneck clusters, we see that they concentrate on different units of each factor. Particularly, bottleneck clusters do not always have bias on the worst units of

TABLE II

THE BOTTLENECKS IDENTIFIED FROM ONE-WEEK QUERY RECORDS. EACH ROW REPRESENTS ONE BOTTLENECK, INCLUDING ITS ID, THE BOTTLENECK CLUSTER AND COMPONENTS. LAST THREE COLUMN ALSO SHOWS THE RELATIVE NUMBER OF FAST AND SLOW QUERIES (#FAST QUERY AND #SLOW QUERY) AND THE FRACTION OF SLOW QUERIES IN THIS BOTTLENECK (%SLOW QUERY). .

ID	Bottleneck Clusters					Bottleneck Components				#Fast query	#Slow query	%Slow query
	CP	NS	$\#image$	$browser$	ad	T_{net}	T_{srv}	T_{dom}	T_{embed}			
B1	*	$worse$	*	*	yes	✓		✓	✓	2.1%	4.6%	68.8%
B2	*	$worse$	$many$	*	*	✓			✓	3.4%	7.0%	67.1%
B3	*	$worse$	*	$U4$	*	✓			✓	3.8%	6.1%	61.6%
B4	*	$worse$	*	*	no	✓			✓	5.2%	8.2%	61.0%
B5	$good$	$worse$	*	*	*	✓			✓	4.1%	5.9%	59.1%
B6	$worse$	*	*	*	*	✓		✓	✓	6.7%	7.8%	53.9%
B7	*	*	$many$	*	yes	✓		✓	✓	10.9%	5.7%	34.5%
B8	*	*	*	$U1$	*	✓		✓	✓	13.6%	5.0%	26.9%
B9	*	*	$many$	*	no	✓			✓	24.7%	8.3%	25.1%
B10	*	*	$many$	$U4$	*	✓			✓	22.8%	6.9%	23.2%
B11	$good$	*	$many$	*	*	✓			✓	23.0%	6.2%	21.1%
B12	*	*	few	*	no	✓	✓		✓	19.4%	5.0%	20.5%
B13	$good$	*	*	$U4$	no	✓			✓	23.5%	4.7%	16.5%
B14	*	$better$	*	*	*		✓		✓	47.4%	5.3%	10.0%

each factors. For example, for $\#image$, many bottlenecks appear in $\#image = many$ rather than $\#image = more$; as for CP, $CP = good$ is more popular in bottlenecks. Since bottleneck clusters intend to find out at least Δ slow queries, we take $\#image$, CP and NS as examples to explain how slow queries are distributed.

TABLE III

THE DISTRIBUTION OF SLOW QUERIES IN EACH FACTOR. THE UNITS ARE DIVIDED BY SPLIT POINTS $s1$, $s2$, AND $s3$ IN FIG. 8. THE UNITS THAT HAS APPEARED IN BOTTLENECK CLUSTERS ARE HIGHLIGHTED.

Factors	Unit1 $[0, s1]$	Unit2 $(s1, s2]$	Unit3 $(s2, s3]$	Unit4 $(s3, \infty)$
$\#image$	$fewer$ 7%	few 27%	$many$ 57%	$more$ 9%
CP	$better$ 11%	$good$ 44%	bad 14%	$worse$ 31%
NS	$better$ 21%	$good$ 18%	bad 8%	$worse$ 53%

Table III gives the distributions of slow queries along the three factor. As expected, the units of each factor, that appear in bottleneck clusters in Table II, also contain a large number of slow queries (highlighted cells). Now the question is why the slow queries are distributed like this. For each unit, we have #slow queries = #query × %slow query. As shown in Fig. 8 (a), (c), and (e), it is true that as $\#image$ increases, and CP and NS become worse, the %slow query rise up. In other words, bad factors can impact SRT more significantly. However, the #slow queries is mainly determined by #query. In Fig. 8, compare the left figures (#query) with the right ones (#slow query), the distributions of slow queries are more similar with that of queries, other than that of the %slow query. This is because the distribution of queries changes sharply, yet the %slow query only varies in relative small range. As a result, slow queries are mainly distributed to the popular units (high #query) other than the poor unit (high %slow query), and so do bottleneck clusters. The split points in Fig. 8 have been introduced before in Section III-A. Note that as CP and NS both have a long tails, their last units (i.e., units of worse) receive a substantial number of slow queries, and also appear in the bottleneck clusters.

In addition, this observation also demonstrate that even for a single factor, where slow queries concentrate cannot be decide directly. It is far more complicated when multiple factors overlap each other.

Observation 2: As for the bottleneck components in Table II, overall, T_{embed}, T_{net} are the two dominated ones. They are the bottleneck components for every bottleneck, except T_{net} missing B14, where the $NS = better$. This indicates that the major constraints for current SRT are the extra time introduced by images, and the network

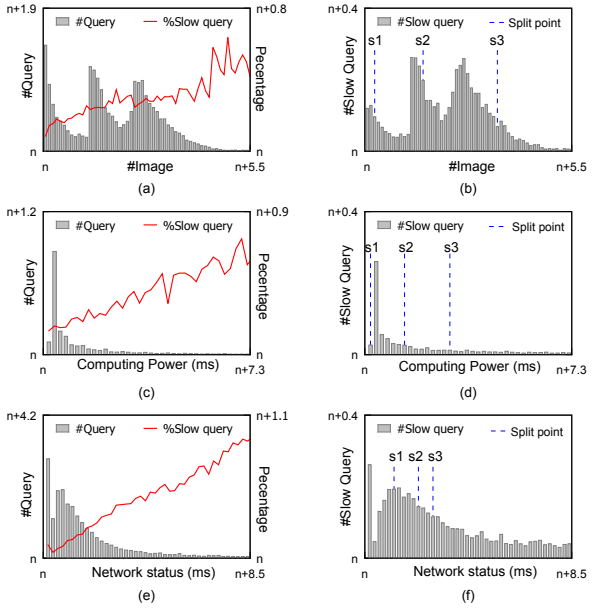

Fig. 8. The distribution of queries, the percentage of slow queries, and slow queries in each factor. The spilt points divide each factor into four units, i.e., $[0, s1]$, $(s1, s2]$, $(s2, s3]$, and $(s3, \infty)$, as described in Section III-A.

transmission time between users and search data centers. One should pay more attention to troubleshoot their causes. In addition to the slow access network of users, the search engine can be responsible for these problem as well. Further debugging is described latter in the case studies. On the other hand, T_{srv} and T_{dom} appear less frequently in the bottleneck components. This implies that for some bottlenecks, they are not critical for the slow SRT.

We also calculate EP of the four components for all the slow query, and their CDF is illustrated in Fig. 9. Focusing on the 80% level, we can get that $EP(T_{embed}) = 83\%$, $EP(T_{net}) = 46\%$, $EP(T_{srv}) = 23\%$, and $EP(T_{dom}) = 14\%$. These results, once again, prove that T_{embed} and T_{net} are the most important reasons for slow SRT. Additionally, an interesting phenomenon is that despite that T_{dom} has the least EP in Fig. 9, T_{dom} appears in 4 bottlenecks (B1, B6, B7, and B8). Whereas, T_{srv} only shows up in 2 bottlenecks (B12 and B14). The reason is that DOM load can be seriously affected by the factors of ad (B1 and B7), worse CP (B6), and browsers of $U1$ (B8), which turns out to be a set of old-fashion browsers. On the other side, server process is basically independent

of all these factors. Therefore, those queries with high $EP(T_{srv})$ are evenly distributed among different clusters, which makes $EP(T_{srv})$ small in each cluster.

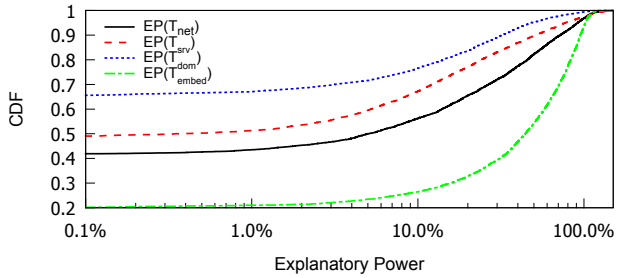

Fig. 9. CDF of explanatory power of different components.

B. Case Study for Debugging Bottlenecks

By leveraging the bottlenecks in Table II, the operators from the search engine further debug along three directions. They finally identify some causes and propose corresponding solutions.

Case 1 (B1-B5): These bottlenecks shows that a plenty of slow queries originate from $NS = worse$. Through further investigating, the operators find out that other than the users' slow access network, the provider's DNS is to blame as well. The DNS is responsible for directing each query to a close data center. However, the DNS policy is not always optimal. The main reason is that the DNS policy updates every two weeks, so that the DNS lacks the ability of choosing best path in the face of realtime events, e.g., ISP network congestions and DDoS attack. To prove this, operators leverage another measurement by a client software of the search provider. The software is used by over 12,000,000 users every day. These users are widely distributed and also use the provider's search engine. Every client downloads a 20KB image from a random search data center every hour. The data center is accessed via IP directly instead of following the DNS direction. As the image costs negligible time of the data center to send out, the download time can be deemed as network transmission time. Operators use this data to generate a speed map of all the search data centers every minute for every province. For a province, the fastest path is called ideal path. They compare the speed of the ideal path with the path suggested by the DNS, and find out that only in 19% of the cases, the path specified by DNS is the optimal. For other cases, the ideal path can yield, on average, 26% faster download from search data centers. So, they believe that a dynamic DNS based on the speed map can, in principal, improve T_{net} of worse NS greatly.

Case 2 (B10): Many embedded images can inflate SRT greatly. An underlying reason is that the browser acquire these images only after the result page is completely received from the server. This problem seems unavoidable at first glance, but after noticing that $U4$ is a set of advanced browsers, the operators come up with a solution, called image pre-fetching, to solve the dilemma. The main idea is that after a query arrived, the server immediately responds the image URL list to the client, in the meanwhile it prepares the result page. By this means, the images can be fetched in advance and loaded into the page when it is arrived. This function is only supported well by $U4$. An offline experiment shows that it can improve T_{embed} by 20%.

Case 3 (B1, B7): These two bottlenecks suggest that ad can increase T_{dom} obviously. Though further debugging, the operators find out that the page with ad usually includes some external resources, which can block the DOM load. A comparison with ad free pages shows that ad can delay T_{dom} by 41% on average. A possible solution is to eliminate the external dependency, and make ad code self-contained.

Case 4 (B8): $U1$ is a set of old-fashion browsers. Besides the common bottleneck components T_{embed} and T_{net}, $U1$ also suffers from long T_{dom}. It is due to both its low performance and the bad compatibility, that hamper page code optimization. When compare with other advanced browsers, $U1$ has 65% longer T_{dom}. Actually, facing such browsers, operators' hands are tied. Although operators have already prepared pages with low overhead for $U1$, but it seems not enough. To avoid $U1$ impacting SRT, operators need to prepare simpler pages with less overhead for $U1$.

Above experience proves the value of FOCUS. The bottlenecks identified by FOCUS can help the operators quickly focus on the right debugging directions, and arrive at the causes of slow SRT.

C. What-if Simulation

Furthermore, we quantify the benefit of using FOCUS via a series of what-if simulations. We simulate the SRT improvements of different solutions, including the ones in the above case studies, that focus on the bottlenecks by FOCUS, and several ad hoc solutions. Some of these ad hoc solutions are proposed by the operators before using FOCUS. By comparing their improvements, we demonstrate that FOCUS is an effective tool for realizing the SRT requirement.

1) Simulation Methodology: Given a solution, we first decide the affected cluster after it is deployed. This is based on the constraints and the characteristic of the solution. For example, the solution of "self-contained ad" in **Case 3** can only affect the queries that trigger ad. Afterwards, in the affected cluster, the solution can improve a particular SRT component according to the problem it intends to solve. For example, "self-contained ad" is designed to eliminate the blocking of DOM load introduced by ad, so it can improve T_{dom}. Last, we divide the overall queries into two parts, one suffers from the problem that the solution aims at, the other do not. We measure the difference of the components from the two parts, e.g., T_{dom} of the pages without ad is 41% shorter than that of the pages with ad. In some cases, we also conduct offline control experiments to measure the difference. We use this measurement as the assumed, or estimated improvement of the solution. For example, according to the measurement, we assume that "self-contained ad" can improve T_{dom} by 41%. All the solutions are simulated on the same query records.

Table IV shows the simulation assumptions for each solution. The first four solutions are designed based on the output of FOCUS, and their details have been described in the prior four case studies. In contrast, we also simulate another five ad hoc solutions without considering the bottlenecks given by FOCUS. Some of these solutions were actually being considered by the operators for deployment in the studied search engine. Their descriptions are as follows.

- *Reduce images*. This solution intends to improve T_{embed} by simply limiting the number of embedded images of a page. Specifically, $\#images = more$ is not allowed. As a result, original queries with $\#images = more$ can be changed to queries with $\#images = many$. The measurement shows that under $\#images = many$, T_{embed} is on average 27% shorter than that under $\#images = more$. Therefore, the assumed improvement of the solution on T_{embed} is 27%.
- *Disable DAT*. About 11% users experience an advanced page effect, called "display as typing", or DAT for short. It automatically exhibits the result pages after each query has been typed. This kind of frequent page loading can bring in high transient computing overhead, and based on the measurement, it can noticeably affect T_{dom} for $CP = worse$, about 38% longer. This solution is turning DAT off for $CP = worse$. The measurement shows that, 5% queries from $CP = worse$

trigger DAT. So the affected cluster of disabling DAT is 5% of $CP = worse$.

- *Complete IP Dictionary*. The DNS direct queries based on an IP dictionary. Unfortunately this dictionary is incomplete. The query, whose IP is not in the dictionary, is directed to a default data center, which can be less than optimal. The measurement proves that the average T_{net} of queries hitting the dictionary is 36% shorter than those missing the dictionary. This solution aims at completing the dictionary. As a result, if a query originally misses the dictionary, we reduce its T_{net} by 36% to simulate the improvement of the complete dictionary. According to the measurement, dictionary miss happens to 9% of $NS = worse$ and 4% of other NS. These are the affected clusters for this solution.
- *Improve CDN cache*. The CDN is an effective infrastructure to decrease T_{embed}. Besides its close location to users, it can further accelerate the download speed of images by caching them. In fact, the cache hit rate is around 90%, and it turns out that the cache miss can add 50% extra time for T_{embed}. So this solution tries to improve the CDN cache hit rate. We assume the ideal situation of no cache miss, which means that 10% queries can yield a 50% reduction in T_{embed}. These 10% queries are randomly selected from the whole queries.
- *Fast ad process*. If a query triggers ad, the server needs a little more time to prepare the content. This solution focuses on decreasing the extra T_{srv} for ad. Our measurement shows that ad can increase T_{srv} by 5% when comparing with ad free queries. As such, the simulation of the solution is to reduce T_{srv} by 5% for queries that trigger ad.

TABLE IV
THE SIMULATED SOLUTIONS AND THEIR ASSUMPTIONS.

Sources	Solutions	Affected Clusters	Improved SRT Comp	Assumed Imprv %
FOCUS	Dynamic DNS	NS=worse	T_{net}	26%
	Image pre-fetching	browser=U4	T_{embed}	20%
	Self-contained ad	ad = yes	T_{dom}	41%
	Simple Page	browser=U1	T_{dom}	65%
Ad hoc	Reduce images	#image=more	T_{embed}	27%
	Disenable DAT	5% of CP=worse	T_{dom}	38%
	Complete IP Dict.	9% of NS=worse 4% of NS=other	T_{net}	36%
	Improve CDN cache	10% of (*)	T_{embed}	50%
	Fast ad process	ad = yes	T_{srv}	5%

To compare the effectiveness of different solutions, we adopt a criterion, called *the percentage of requirement completion*. As described earlier, another $\Delta = 4.6\%$ slow queries need to be accelerated to satisfy the requirement. In the simulation, solution i can accelerate N_i slow queries. Therefore, we define the percentage of completion of solution i as N_i/Δ. It indicates how much the requirement has been achieved by solution i, and can be used to measure the effectiveness of solution i. We conduct two types of simulations, one is simulating the individual deployment of each solution, and the other is for combined solutions.

2) Results of Individual Solution: The individual result of each solution is shown in Fig. 10. Overall, we see that the first four solutions, designed according to FOCUS, yield a relative high percentages of requirement completion, ranging from 25% to 36%. On the other hand, the other five ad hoc solutions achieve much less, only from 1% to 12%. The reasons are two folds:

The first one is whether the affected clusters are the bottleneck clusters. In Table IV, we see that the affected clusters of the first four solutions are the bottleneck clusters, which ensure at least Δ slow queries. Conversely, the affected clusters of the ad hoc solutions are not. In particular, solutions of "Reduce images" is deployed on $\#image = more$, which is not a bottleneck cluster. As a result, though "Reduce images" and "Image pre-fetching" are both designed to improve T_{embed}, and the former even has a higher improving percentage, the results in Fig. 10 shows that "Reduce images" is much less effective than "Image pre-fetching". This demonstrates the importance of bottleneck clusters for improving SRT. Additionally, "Disable DAT", "Complete IP Dict.", and "Improve CDN cache" only affect a small part of the bottleneck clusters, they thus cannot accelerate too many slow queries either.

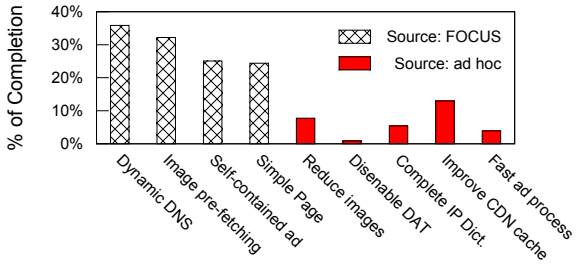

Fig. 10. Individual simulation result of each solution. Each bar represents the percentage of requirement completion of a solution.

The second reason is whether the improved components is the bottleneck components. We see that "Fast ad process" is deployed for $ad = yes$, which indeed covers bottleneck clusters. Whereas, it tries to improve T_{srv}, which is not in the bottleneck components. This indicates that T_{srv} only accounts for a small part of slow SRT. The point is further verified by the real measurement. It shows that the T_{srv} with and without ad only differs by 5%, and improving T_{srv} is thus less worthy here. On the contrary, "Self-contained ad" is deployed on the same cluster, but it targets at T_{dom}, which turns out to be a bottleneck component. As expected, the measurement shows a high improving percentage for "Self-contained ad", i.e., 41%. Finally, "Self-contained ad" performs much better than "Fast ad process" in the simulation.

3) Results of Combined Solutions: Although the solutions from FOCUS present much more effectiveness than the ad hoc ones, the results in Fig. 10 suggest that they cannot realize the requirement individually. This is because even in the same bottleneck cluster, the slow queries are caused by different SRT components. As such, a solution, improving only one component, can hardly accelerate all the slow queries in the bottleneck cluster.

In practice, many solutions are deployed simultaneously to improve SRT. Therefore, we also simulate the effectiveness of combined solutions. As the improvements of those ad hoc solutions are relative small, we only simulate the combinations of solutions from FOCUS, i.e., the first four solutions in Table IV. We evaluate the situation of all the four solutions deployed simultaneously, as well as the situations of any three of them working together. The results are shown in Fig. 11. For the case of four solutions deployed together, the percentage of requirement completion is 113%, implying that the requirement has been better achieved. If three of them are deployed, the requirement cannot be fully satisfied, but the results are still reasonably good, ranging from 80% to 92%. When comparing this result to those ad hoc solutions in Fig. 10, we find that requirement completion of FOCUS based solutions is four times higher than those ad hoc ones.

In summary, without the outcome of FOCUS, one might come up with any solutions. At first glance, they all seem like promising. Yet, from the simulation results, we see that the solutions, focusing on the bottlenecks, can work more effectively.

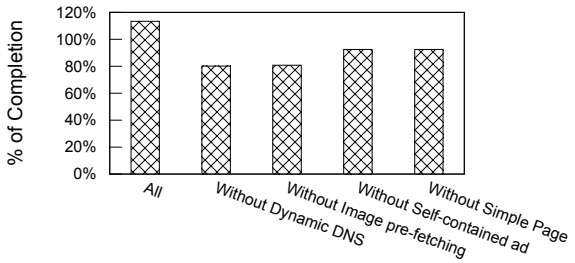

Fig. 11. Simulation results of combined solutions (from FOCUS). The bar of "All" is the result of all the four solutions combined together. The other four bars represent the results of combining any three of the four solutions, where the label of X axis indicates the omitted one.

V. Related Work

Web acceleration. Many efforts have been put into the improvement of web page load, which can also benefit SRT, such as TCP fast open [16], TCP timeout mitigation [5], host name pre-resolving and TCP pre-connecting [6], [7], CDN technologies [8], [9] and front-end architecture [10], [11]. While these approaches can accelerate some aspects of SRT, our work focus on revealing the bottlenecks of slow SRT and help operators debug the problem. We argue this job is the step before deploy any acceleration methods.

Web performance analysis. To help locate reasons of slow web page, some researchers have built elaborate tools. For example, [17], [18] leverage browser instrumenting techniques to demystify the page objects loading dependency. [19] seeks the bottleneck of page load time from the home gateway. Also, there are many other tools such as online services [20], [21] and browser plugins [22], [23] that can help providers debug the page load time. However, neither these active measurements can deploy at a large scale, nor they can reveal the experience of real users. As a result, they are not feasible for debugging slow SRT from the perspective of search providers.

SRT and debugging. [2] conducts a research for understanding SRT variations from the provider side view. They attempt to diagnose the significant change of SRT, but their empirical decision logic is not general. Many papers performs debugging in network [24], [25] and ad system [15]. The solution of [15] inspirits our definition of bottleneck components. Besides, machine learning is a general and effective tool. Especially unsupervised learning (e.g., k-means and DBSCAN [26]) can identify patterns of data automatically and seems feasible for our problem. However, they involve several obstacles. In addition to the drawback of specifying intrinsic parameters, the clusters they identified do not have clear boundary along each dimension and are usually represented by their centroid and deviation level, thus difficult to interpret. Instead, we take advantage of another approach, i.e., hierarchical heavy hitter(HHH) [12], which can lead to more meaningful results.

VI. Conclusion

Search response time (SRT) is a very critical KPI for search engines. Milliseconds increase in SRT can lead to millions of searches lost. In order to improve SRT, debugging slow SRT is indispensable as the first step. But so far, it has not been explored very much. In this paper, we propose FOCUS, a systematic framework to automatically identify bottlenecks of slow SRT. We collect diverse impact factors and SRT components, and use HHH and explanatory power to pinpoint the bottlenecks. With FOCUS narrowing down the debugging space, operators can further investigate the root causes of slow SRT in a more specific direction. We deploy FOCUS in one of global top search engines. The bottlenecks provided by FOCUS successfully help the operators locate four main causes of slow SRT, which, without FOCUS, would have been the needles in a haystack. Our what-if simulation also proves that, for the given SRT requirement, the solutions focusing on those bottlenecks are much more effective than ad hoc ones.

As far as we know, FOCUS is the first work towards systematically debugging slow SRT. Since both the data collection and analysis methodology of FOCUS are general, FOCUS can be easily applied to other search engines as well.

Acknowledgement

This work was supported by the National Natural Science Foundation of China (NSFC) under Grant No. 61472214, the National Key Basic Research Program of China (973 program) under Grant No. 2013CB329105, the State Key Program of National Science of China under Grant No. 61233007, the National Natural Science Foundation of China (NSFC) under Grant No. 61472210, the National High Technology Development Program of China (863 program) under Grant No. 2013AA013302.

References

[1] Search engine. http://googleresearch.blogspot.com/2009/06/speed-matters.html/.
[2] Yingying Chen, Ratul Mahajan, Baskar Sridharan, and Zhi-Li Zhang. A provider-side view of web search response time. In *SIGCOMM 2013*.
[3] Speed matters. http://www.sportrichlist.com/top10/best-search-engines-in-the-world/.
[4] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Conference Talk*, 2009.
[5] Tobias F., Nandita D., et al. Reducing web latency: the virtue of gentle aggression. In *SIGCOMM 2013*. ACM.
[6] E. Cohen and H. Kaplan. Prefetching the means for document transfer: a new approach for reducing web latency. In *INFOCOM 2000*.
[7] Srikanth S., Nazanin M., Nick F., and Renata T. Accelerating last-mile web performance with popularity-based prefetching. *CCR*, 42(4), 2012.
[8] Ao-Jan Su, David R Choffnes, Aleksandar Kuzmanovic, et al. Drafting behind akamai (travelocity-based detouring). *CCR 2006*.
[9] Ao-Jan S. and A. Kuzmanovic. Thinning akamai. In *IMC 2008*.
[10] Yingying Chen, Sourabh Jain, Vijay Kumar Adhikari, and Zhi-Li Zhang. Characterizing roles of front-end servers in end-to-end performance of dynamic content distribution. In *SIGCOMM 2011*.
[11] Adam Lazur. Building a billion user load balancer. In *Velocity Web Perfomance and Operations Conference*. O'REILLY, 2013.
[12] Cristian E., Stefan S., and George V. Automatically inferring patterns of resource consumption in network traffic. In *SIGCOMM 2003*.
[13] http://www.nngroup.com/articles/response-times-3-important-limits/.
[14] Pagespeed. https://developers.google.com/speed/docs/insights/mobile.
[15] Ranjita B., Rahul K., Ramachandran R., George V., Surjyakanta M., Hemanth M., and Piyush S. Adtributor: revenue debugging in advertising systems. In *NSDI 14*.
[16] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. Tcp fast open. In *CoNext 2011*.
[17] Xiao Sophia W., Aruna B., Arvind K., and David W. Demystifying page load performance with wprof. In *NSDI 2013*.
[18] Zhichun Li, Ming Zhang, Zhaosheng Zhu, et al. Webprophet: Automating performance prediction for web services. In *NSDI 2010*.
[19] Srikanth S., Nick F., T., et al. Measuring and mitigating web performance bottlenecks in broadband access networks. In *IMC 2013*.
[20] Google page speed. https://developers.google.com/speed/pagespeed/.
[21] Web page test. http://www.webpagetest.org/.
[22] Firebug. getfirebug.com/.
[23] Chrome developer tools. https://developer.chrome.com/devtools.
[24] He Yan, Lee Breslau, Zihui Ge, Daniel Massey, Dan Pei, and Jennifer Yates. G-rca: a generic root cause analysis platform for service quality management in large ip networks. *Networking, Transactions on*, 2012.
[25] Paramvir B. et al. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *CCR 2007*. ACM.
[26] Martin Ester, Hans-Peter Kriegel, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1996.