All-Spark: Using Simulation Tests Directly in Production Environments to Detect System Bottlenecks in Large-Scale Systems

Jialiang Lin
Alibaba-inc
Hangzhou
jialiang.linjl@alibaba-inc.com

Jun Zhang
Alibaba-inc
Hangzhou
youji.zj@alibaba-inc.com

Yu Ding
Alibaba-inc
Hangzhou
shutong.dy@alibaba-inc.com

Liping Zhang
Alibaba-inc
Hangzhou
liping.z@alibaba-inc.com

Yin Han
Alibaba-inc
Hangzhou
hanyin.hy@alibaba-inc.com

ABSTRACT

With the rapid growth in e-commerce, large-scale promotional activities have become a popular concept. However, when the existing system cannot be adjusted efficiently to adapt to the tremendous traffic in the promotion period, which is hundreds of times more than the volume on normal days, it becomes a bottleneck that restricts the continuous growth of the online business. Traditional capacity prediction methods have been proven to be incapable of making accurate predictions for such special scenarios, because of a variety of unpredictable system bottlenecks. Simulation testing in a completely new test environment for such a large scale has a number of defects and limitations, such as the high cost of setting up the environment and the difficulty of testing the entire environment. Moreover, bottle-necks found in the test server may be different from those in the production server. We investigated online simulations in the production environment and built a complete simulation test system called All-Sparks. This solution solved a long-standing problem of simulation testing with large traffic in the production environment without causing any data pollution. The simulation test revealed hundreds of bottlenecks under a high workload pressure every year to eliminate the hidden problems caused by new applications. The final capacity evaluation result was deviated by less than 5% from the actual capacity, and the error rate was small (<2%); both of these are significant improvements over the traditional prediction results. This solution also provided a framework with good expansibility to multiple scenarios other than stress testing.

CCS CONCEPTS

• **Computer systems organization** → **Reliability; Availability; • General and reference** → **Reliability;**

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Middleware '18, December 10–14, 2018, Rennes, France

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5702-9/18/12...\$15.00

<https://doi.org/10.1145/3274808.3274809>

KEYWORDS

ACM proceedings, $\LaTeX$, text tagging

ACM Reference Format:

Jialiang Lin, Jun Zhang, Yu Ding, Liping Zhang, and Yin Han. 2018. All-Spark: Using Simulation Tests Directly in Production Environments to Detect System Bottlenecks in Large-Scale Systems. In *19th International Middleware Conference (Middleware '18)*, December 10–14, 2018, Rennes, France, Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3274808.3274809>

1 INTRODUCTION

1.1 Business background

Starting from 2009, our company has held a big online promotion event every month and a large-scale online promotion event every year; thus far, all of these events have been successful. However, the challenges to the backend system increased quickly and have become a considerable problem. In the early days, we could use traditional methods to predict capacity (number of machines needed) and to add machines urgently to handle the additional volume on the big promotion days which is a special day with a considerably large (more than a dozen times) volume increase.

The business increased exponentially in 2010, and the peak-time transactions per second (TPS) doubled that year. Order submissions, the most important scenario, increased from 40,000 TPS to 80,000 TPS. Although such a huge increase was predicted and additional machines were prepared, an unexpected severe problem occurred on the big promotion day, when two important and high-resource-consuming applications had their virtual machines deployed on the same physical machine, and a contention occurred in peak time, which dragged the system down.

We then realized that we had entered a stage where the traditional capacity prediction method might not work well. Thus, we had to re-evaluate the capacity prediction and develop a new method to meet the business needs.

1.2 Challenges

Here, we summarize three special characteristics of our big promotion:

- **Huge volume difference:** The number of requests during a big promotion is hundreds of times greater than on a normal day. In 2017, the number of order submissions, the most complicated scenario, was observed to be 330,000 TPS during the peak time, while it was only 3,000 on a normal day.
- **Big business model and environment change:** According to our statistical analysis, on a normal day, a single user purchases 1.5 pieces of goods on average. This number jumps to 5 on a big promotion day and causes a considerable difference in handling the purchase logic, such as browsing goods and coupon usage. To handle the high volume, we need to increase the number of servers by 2 to 3 times for processing. Moreover, we need to borrow these servers only a few hours before the big promotion and release them right after, for cost savings.
- **Low tolerance of error rate:** In an e-commerce scenario, a high error rate indicates less success in making order, which directly results in a business loss.

Before 2012, we used traditional methods to prepare for a big promotion: we calculated the performance baseline of a single machine, predicted the volume on the big promotion day, and then calculated the required machine capacity. However, we found that in addition to the lack of machines, system bottlenecks are a key factor affecting performance. A bottleneck can be anything: a shared object, a wrong configuration, or database connection numbers. The more complex a system is, the harder it is to find all the system bottlenecks. Our research revealed a strong correlation between the bottlenecks and the complexity of the entire environment. When the environment changed, such as NIC failure on a physical machine or incorrect configuration of network load balance(NLB), the bottleneck changed as well. Moreover, we found that more than 90% of the bottlenecks only occur when the system is under high pressure and will not surface on a normal day.

Therefore, we concluded that a theoretical prediction will not help us to identify bottlenecks and, to accurately predict the number of additional machines required for the big promotion. We also need to conduct simulation testing to verify and rectify the capacity predicted; to serve this purpose, we developed the All-Spark system.

1.3 Related work

A number of studies([5],[6],[10]) have been conducted on load testing and stress testing in the field of capacity planning and bottleneck detection.

The first task is to determine the test environment. We chose to conduct the test in the production environment directly, because our big promotion requires more than tens of thousands of servers, which makes it very expensive to set up a new test environment from scratch.

However, to run simulation test on production environment, there is a risk that the actual production data might get polluted and the dirty data issue must be solved. It is important to find a way to quarantine the test data from the actual data to ensure zero pollution to the production environment. Thus far, no solution has been developed to prevent pollution in an online test of such a large scale.

Kraken([16]), introduced an approach to redirect a real request to multiple specified servers and ensure that the stress test received the maximum workload and then calculate the cluster workload or bottleneck. A considerable advantage of this method is that no dirty data is generated. However, the limitation of Kraken's method is that it can only detect bottlenecks in relatively stable systems. In our case, the number of requests on a normal day was considerably lower than the big promotion volume and might not be sufficient for a "stress testing" on the selected servers.

Besides the stress testing, some new technologies such as real-time scaling up have been proposed([13],[14],[17],[15],[11] and [12]). When the monitoring system detects the capacity or when the performance reaches the threshold/bottleneck, new servers can be promoted in a few seconds by using the modern container technology.

However, although the server can be prepared very quickly, the applications running on it still need several minutes to finish the start, registration, and publish processes. This implies that during this period, new requests exceeding the system's capacity can-not be handled. This is unacceptable for a commercial business since user cannot place his/her order for several minutes continuously.

Because of the special business needs, we developed a new system called All-Spark, which can be used for online simulation testing in the production environment, with a stress test volume as high as the big promotion level, without any pollution of the business data.

1.4 Achievements

Stress testing with a large volume of data in the production environment is a classic topic discussed for decades without a common solution. We used a combination of external requests, simulation tags, and middleware technology to build the All-Spark system, a traffic simulation system. When All-Spark was launched in 2013, we found 600 bottlenecks that we were not aware of earlier. Although the peak volume for the "create order" scenario, which is one of the most complex scenarios, increased from 40,000 TPS to 330,000 TPS in 2017, All-Sparks accurately evaluated the required capacity and identified the bottlenecks during the simulation test, to ensure no downtime during the big promotion. In the past few years, All-Sparks has been developed quickly as a complete system, for accurate capacity evaluation with 2% deviation (of the real capacity by a postmortem analysis), and an error rate of within 2%, to support big promotions for five years continuously.

Moreover, this system is not restricted to simulation testing only. With its kernel concept for successful data and process logic separation, it can be extended to multiple other scenarios, as discussed in Section-5.

2 SYSTEM DESIGN

2.1 Capacity Planning

2.1.1 Prediction of TPS.

The TPS for each scenario is predicted by our large-data department on the basis of historical data and the current year's business increase. They provide the minimum number of requests for different scenarios. We need to calculate the TPS of the backend services. Figure 1 shows a sample topology for different scenarios.

Figure 1: Topology generated by trace module

Figure 2: Gap between actual peak TPS and predicted TPS

From the above topology, we know that the TPS of scenario-a is T_a , and each request of scenario-a will call service-2(S_2) for $C_{1,2}$ times and service-3(S_3) for $C_{1,3}$ times, which $C_{1,2}, C_{1,3}$ are generated by trace module described in Section 2.4; Then the TPS of service-2 should be $S_2 = T_a * C_{1,2}$. Besides, S_2 will also call S_3 for $C_{2,3}$ times. Then the TPS of Service 3 will be $S_3 = T_a * C_{1,3} + S_2 * C_{2,3}$. And we can calculate the TPS of all services similarly.

In the programming implementation, we started from the entrance node and calculated the number of nodes in the next level on the basis of the outgoing link. Then, we performed recursion to move down, until all the nodes were calculated.

In the postmortem analysis after a big promotion, we found that this method could predict a request within 5% difference with the real data on a big promotion day. This also showed the accuracy of Trace Module.

Figure 2 shows the predicted volume request versus the real volume request for services in a certain scenario, with the gap listed in percentage.

2.1.2 Theoretical capacity planning.

We have tried multiple modern capacity prediction solutions, however, none of them can put all the potential bottlenecks into considerations. In our case, different services will use different ways to do capacity planning. The most common way is to calculate the minimum number of servers needed via dividing the prediction

TPS by the baseline of this service. After adding some buffer based on our experiences, we get the predicted capacity.

According to our experiment, if all the bottlenecks are detected and fixed, the actual capacity should be close to this theoretical value.

The creation of simulation requests is a challenge mainly because of the frequent changes in the system interface and the difficulty of simulating real-world requests.

After repeated tests and comparisons, we finally found a solution to solve both problems: sending simulation requests from the client side, instead of sending them from the subsystems, as the internal function calls across the systems is relatively easy to change, while the outside interface is considerably stable. Moreover, request simulation from the out-side is similar to the real-world scenario (99.999999% buyers place orders from outside of the CorpNet). Thus, we developed a complete test to cover all the links on the traffic chain.

With this solution, we had to take care of two related problems:

- Since the simulation was conducted directly in the real production environment, the simulated data could pollute the production environment; therefore, we had to quarantine "dirty data".
- To reduce this impact to the real users, the simulation time had to be well controlled

Furthermore, in order to identify and efficiently fix the problem revealed by the simulation test, we needed a strong tracing system to track the system running status in considerable detail and quickly narrow down the problematic component.

Compared with the complexity and instability of the internal logic in a large e-commerce system, the URLs for the clients to visit the web portal are simple and stable. For example, for a browsing scenario, the sample URL is <https://<domainname>/item.htm?id=36952255601> and for a purchasing scenario, the sample URL is <https://<domainname>/order.htm?id=36952255601&count=2>. Here, "order.htm" indicates the "place order" request, "id" represents the identity of the goods, "count" denotes the purchase number, and the request header contains the user's log-in session ID, which includes the user ID and other authentication information.

Upon receiving this request, the backend server needs to collect/-calculate the related information, such as seller, discount, and stock. These items are called the basic business factors and are the crucial information that affects the response. Therefore, we extracted the basic business factor data from the production database with the same data size, filtered and desensitized the sensitive fields, and then stored the processed data into a database for the simulation testing. These data were called "shadow data" and the database was called "shadow database."

Furthermore, certain requests require state; e.g., a request to purchase goods needs to the user's log-in session information. For these requests, we also needed to simulate the log-in request, obtain the session information from the log-in reply, and then insert it into the buy request header.

Finally, we added a flag in the request header, indicating whether the request was a simulation request or not, along with a long bizId to indicate the request scenario. Thus, we obtained Table-1:

Table 1: Fields in URL

Fields	Example
Request URL	http://<domain>/item.htm?id=36952255601
token	3d74b50b3a35f
isSimulation	True (mark whether it is a simulation request)
bizId	A unique long parameter to determine the scenario

Figure 3: Process for sending simulation requests

By using this approach, we could easily generate a simulation request and create/maintain the shadow database (by synchronizing with the product data-base before the simulation test). Therefore, in most of the scenarios, the internal function change within the system did not affect the simulation request using an external URL. Further, when the external URL changed, we could quickly adapt to the change by modifying the URL and bizId.

2.2 Simulation request engine

Unlike the traditional simulation tests which mostly run on a normal PC in CorpNet, our test engine was placed on an external Content Delivery Network(CDN,[2]) node. This was because the end users' requests, which were distributed across the nation, first reached the closest CDN and were then forwarded to the CorpNet gateway. Therefore, a request was sent from the CDN instead of CorpNet to test the complete traffic chain in a real-world scenario.

We first generated the simulation requests from the shadow database, sorted the requests on the basis of the state information (such as session ID), and stored the results in files. Next, we uploaded these files for the engines deployed in the CDN node to read from. When the simulation test started, the engines read the file, sent simulation requests to the CDN in the predefined order (to simulate end-user actions), and then let the CDN forward the request to the CorpNet gateway.

In this approach, a single engine can provide 80,000 transaction per second(tps); therefore, we could easily simulate 50,000,000 TPS with hundreds of engines configured on the CDN and multiple

protocols supported at the same time, including http, https, spdy, and ACCS([1]).

Following are the common problems which might occur during sending simulation requests:

- **Session expiry:** This is usually caused by an improper sending order. For example, order creation requires the user's log-in session information, which can expire if the simulation test runs for a long time. This can be solved by regularly refreshing the log-in session and letting CDN engine get the latest session information when creating a simulation request.
- **Traffic control:** The application module, besides the gateway, can also have a traffic control mechanism and block the incoming traffic when the traffic volume exceeds the predefined threshold. Therefore, we need to increase the threshold on these modules or even disable the traffic control function to ensure that the system performance can be fully tested.
- **Business logic:** This is the most common reason blocking a simulation test. For example, when a "buy request" is submitted, the expected result is successful order creation. If HTTP 200 is returned with an error code such as "no discount", the ongoing simulation cannot be processed. Therefore, this is not a successful result, and we need to engage the application owner to check why the error code was returned.
- **Application performance:** During the simulation test, if a high response time (RT) is observed, an error such as the HTTP 500 error usually occurs on a single application module on the request chain. Similar to the business logic issue, we need the application owner to identify the system bottleneck and eliminate the problem.

2.3 Simulation requests with almost zero intrusion

Figure 4 shows the sample workflow for how a user request is handled by the system. After node-A receives a user request, it sends an RPC call to node-B. Node-B gets the call and queries the data in remote cache1, which needs to call the DB engine to query the DB records. The result is returned to node-A after being calculated on node-B. Then, node-A sends another RPC call to node-C, which calls node-D. Unlike the other nodes, node-D needs to call a third-party node to get the data. Finally, node-A returns the result from node-C to the end user and stores it in remote cache2 for future use.

There are three types of nodes in this sample work-flow:

- (1) **Footprintless node.** This node only consumes resources (such as CPU) when handling a request but does not leave any trace (such as cache data) in the node. Nodes-A, B, C, and D are all footprintless nodes responsible for the calculations and the passes; they do not care about whether the request is genuine or simulated.
- (2) **Internal footprint node.** The requests leave data on these nodes, such as a new record updated in the database and the event log, or a query result temporarily stored in the cache. This is where the so-called "dirty data" comes from.
- (3) **External node.** Unlike the two above mentioned nodes, this node needs to call a third-party system when handling a

Figure 4: Sample workflow for how a user request is handled by the system

request. Therefore, if the simulation requests are really forwarded to third-party systems, they will have to bear unnecessary workload because of the invalid data input. For example, when simulating a pay request, we need to call a third-party pay system; however, when there is no such account in the third-party system for the simulated user, such a request will consume resources in the third-party system but will finally return a failure result.

Therefore, during the simulation test, the footprint-less node is not affected (not aware about the simulation), We only need to care about the footprint nodes and external nodes, to enable them to handle simulation requests without polluting the business data, which is discussed in Section ?? . However, before doing so, we need to have an efficient method to pass the simulation flag to each affected node, this is described in Section 2.3.2.

2.3.1 Passing simulation flag across nodes.

When the web framework receives a request, it will first check the `isSimulation` flag to determine whether this is a simulation request. If yes, it will generate a `requestId` (for tracing purposes) and create the following small data structure, which only uses 30+ bytes and is easy to copy.

```

TracingContext{
    byte isSimulation;
    long bizId;
    long requestId;
    {Param1,param2,...}
}

```

With this data structure, the workflow is as follows:

- (1) The web framework adds this data structure into the thread context of the processing thread for this request.
- (2) When an RPC call is generated, the RPC framework fetches this data structure from the thread context, inserts it into the

RPC protocol (request body), and passes to the downstream node.

- (3) The downstream node receives the RPC call, extracts the data structure from the RPC request body, and inserts it into the thread context of the processing thread on the downstream node.
- (4) Similarly, in an asynchronous call framework, such a message can take same action to pass the tracing context data structure.
- (5) If it is a call for the cache or the database, the remote cache API or JDBC engine can be used to reach the same result.

2.3.2 Simulation quarantine on middleware.

As discussed in the introduction of Section 2.3, we need the footprint node and the external node to handle a simulation request without pollution. However, we do not want to make too many code changes in each node only for test purposes (there could be millions of nodes in the environment, as each server can have dozens of nodes running on it). We figured out a solution to make code changes only on the middleware in order to distinguish these requests, which resulted in considerably few code changes. There are about 10 types of middleware modules, and thousands of instances for each type with the exact same code. We only needed to update 10 types of middleware modules and then deploy the changes on all the middleware instances. The footprintless node is unaware of the request type (genuine or simulation); therefore, the middleware of such code can be left unchanged. If we want to remove the footprintless node from the test, we can make the middleware (such as an RPC framework) return a mock reply; thus, the simulation request will not reach the footprintless node, and the node will be excluded from the simulation test.

The internal footprint node needs to store the data generated during the simulation. In order to not pollute the production environment, we can use a shadow area (shadow database or shadow cache, as discussed in Section 2.2) to only store the simulated data; the production area (production database and production cache) will be used to store genuine data. Therefore, in the middleware before the footprint node, if the request is identified to be a genuine request, it will be redirected to the production area for further processing; and if the request is identified to be a simulation request, it will be redirected to the shadow area.

A special scenario occurs when the request contains both genuine data and shadow data; e.g., a genuine user tries to browse shadow goods by manually inputting the goods ID in the URL. For security purposes, such a request will be treated as invalid on the middleware, to prevent malicious attacks by using the simulation flag.

For third party requests, the external node simulates the average response time of a request and returns a mock result; therefore, the downstream third party will not be affected.

With these three different processes, the workflow changed as shown in Figure 5 ; thus, all the simulation requests and genuine requests can be separated.

Additionally, a switch is added to the middleware. When it detects that the node is in an unstable status, e.g., the CPU utility exceeds the threshold, it will discard all the incoming simulation

Figure 5: Revised workflow. Green dashed line shows the path of a simulation request; blue solid line shows the path of a genuine request

requests by returning a mock reply. Thus, the simulation requests will not further consume the node resources, and the genuine requests can still work normally.

In the realworld implementation, we deploy a consistent framework such as Dubbo([4]), Rocket MQ([8]), and Tengine([9]), and use the RPC, DB Engine, Message, and HTTP protocols to connect these nodes. As mentioned early in this section, the processing logic (code changes in the middleware module) is controlled in the framework and is transparent to the existing business nodes. Figure 6 shows an example of an additional workflow added to the middleware; it does not change business nodes at all.

Therefore, by using the simulation quarantine on the middleware, we can run the simulation test in the production environment directly, with zero pollution to the production data and there is very little intrusion into the production system.

2.4 Trace module

A simple request may be served by hundreds of applications, which consist of thousands of machines, and this request may be handled randomly by any of these machines. Once an error is observed in the simulation test, we need an efficient method to identify the culprit node on the application chain.

We concatenate the request id and the trace id (in [?]) from all the nodes to generate a trace report of the chain. We record the requestId, bizId, node entry time, and node exit time in the monitoring log, use agents to collect all of these logs from different servers, and then aggregate this information with the same requestId and bizId to generate a calling chain. We then check the error code and Response Time(RT) to easily identify which scenario (node) caused the problem. Figure 8 shows an example chain. With this chain, we can easily locate the node that consumes most of the time and the node that returned an error.

Figure 6: Workflow of a framework

Figure 7: Tracing modules

2.5 Summary

We combined all the above mentioned ideas and designed the system structure of All-Spark as shown in Figure 9:

We first built a shadow database with the basic factors and generated a simulation request with the same amount of big promotion data and stored the result in the file. The simulation agent in CDN reads the file and sends the simulation requests to the company gateway. The application middleware checks the request, redirect-s it to either the production area or the shadow area for further

Figure 8: Calling chain

Figure 9: All-Spark structure

processing, which naturally quarantines the simulation data from the genuine business data. Finally, the monitoring agent collects the log file, builds the call chain, and generates a tracing report to facilitate the identification of the problematic node.

3 SIMULATION TEST

Before the big promotion, we try to predict the number of additional machines required. There are many different prediction methods, and we will not discuss them in detail as they are not the focus of this study. Furthermore, as mentioned in Section 1.2, the prediction result is a theoretical number and cannot cover all aspects, such as unexpected performance bottlenecks, undiscovered business logic errors, or unexpected configuration failures. All these issues need to be identified in the simulation test on the basis of theoretical predictions. If these problems cannot be revealed and fixed in the simulation test, then they could occur on the day of the big promotion, increase Response time, or cause purchase failures, thereby considerably impacting the user experience.

We usually start the simulation test 20 days before a big promotion, to leave sufficient buffer time to conduct 4 to 5 rounds of tests to solve all the problems observed. To minimize the potential

Figure 10: Two phases in simulation test

impact to the real business (such as the workload on footprintless nodes), we choose the volume-rough hours (1 AM to 4 AM) to launch the simulation test.

Each round of tests has three stages: environment setup, simulation test, and environment recovery. The time required for each test ranges from 30 min to 3 h, depending on the problem observed. It is quite common to use the entire 3-h duration and still have some problems unresolved. In such a case, we record the system status, restore the production environment, and continue the test the following day. This will be further discussed in Section 3.2.

3.1 Environment setup

Sometimes, the postmortem analyses reveal that the number of servers required on the day of a big promotion is 2.5 to 3 times more than that required on a normal day. For saving costs, we do not retain the additional machines on normal days and only borrow machines and build this special environment a few hours before the big promotion day or right before a simulation test.

Before 2012, we used virtual machines for the deployment, which took 4 h to increase the number of machines from 20,000 to 50,000. In recent years, we started using the mature container technology, such as pouchcontainer([7]),docker ([3]), which considerably reduced the deployment time. As mentioned in Section 1.3, the deployment can be completed within seconds in theory. However, the application running on the server takes an additional 1 to 2 min to start, including registration and publishing. This leads to a bad user experience during this period. Therefore, we need to control the requirement for realtime expansion to as little as possible. Furthermore, if the number of machines to be expanded is smaller than the maximum capacity of the service discovery, e.g., 5,000 service instances for one time, then the deployment time is approximately the same. However, if the number of services exceeds the limitation and additional RPC service processes, such as service publish, subscribe, etc. are required. This makes the time needed increases linearly, depending on the capacity of these procedure. Currently, we use the most up-to-date technologies and can deploy 200+ applications on several hundreds of thousands of servers in 40 min.

3.2 Simulation test

3.2.1 Send simulation requests.

The entire simulation test is divided into two phases, as Figure-10 shows, phase 1 is for the sharp rise (cold startup), and phase 2 is for the consistent high pressure.

Phase 1 is to simulate the exact scenario of 00:00 AM on the day of a big promotion. The volume of requests increases from the daily level to the promotion level, which is hundreds of times larger, only in 11C2 s. Based on our experience, 12% of the bottlenecks occurs during this phase. Because the system has been idle for a long time and need time to warm up, for example, to load cold data from the disc into memory, to instantiate the first called class, to set up DB connections, etc. All these problems are not easy identified by code review, but in this case it is easy to expose.

Phase 2 is to maintain a large request volume for 10 to 30 min; therefore, the application system will be under high pressure for a relatively long time, and the application owner can then check for the business logic issue and the performance bottleneck if any unexpected problem is observed.

3.2.2 Capacity fine-tuning and bottleneck detection.

We define two types of counters on the basis of the business needs. One type is application counter, represented by "error rate". If the application is unable to process the requests as expected, an error code will be returned. Most of them are caused by business logic, for example, when a user orders with a coupon, and it is expected that user will successfully make this order. However, the system could not read the coupon info as the cache stored the coupon info was down due to the high workload and could not return the coupon info. Thus, a "Fail to make order as no available discount" is returned instead of "make order successfully". Such error will seriously impact the user's experience and should be solved immediately. The error rate is the ratio between the number of error codes returned and the number of total requests, and the rate should be lower than 2%. The other type is machine performance counter, including response time (RT), CPU utility, memory usage, network bandwidth, and disk I/O.

If the error rate is increased and exceeds the threshold (we set 2% as the threshold on the basis of the real user experience), we assume that there is a serious problem. Therefore, we need to start the investigation immediately until the root cause is identified and the error rate drops to the normal level. The machine's performance counter is acceptable if the performance data bounces up and down or even exceed the threshold for a brief period. If the performance data can be auto-recovered and does not exceed the threshold repeatedly, no specific action is needed.

When further investigation is needed, we can utilize the tracing module discussed in Section 2.3, to quickly identify which application causes a problem in the entire chain and then involve the application owner to troubleshoot the problem. We will discuss bottleneck and capacity issues in detail in Sections 4.2 and 4.3.

When the system is stable over the watch window (30 min), with the error rate and the performance counter under the threshold, we can consider. Once the system has reached a stable state, we will fine tune the capacity based on machine performance counters. For example, if the average cpu usage is less than 50%, the capacity is considered to be larger than the minimum capacity, we will reduce the number of machines and vice versa. The performance counter is determined by the application owner because different applications have different bars.

Minimum capacity is the number of minimum machines needed in boundary condition during simulation test. It can be used as

standard for all capacity planning. Before the minimum capacity is tested out, the stable result during watch window is treated as the capacity planning for All-Spark, and will be used as the reference to prepare machines before big promotion event. This is proven to be quite successful: after using the capacity planning result from All-Spark simulation test, we did not run into resource shortage situation during big promotion, and the machines prepared at the beginning can stably handle requests until the end of event.

Based on our experience, the system does not reach a stable state, for example, the error rate is still higher than 2%, when the simulation test window is close to the end and issues haven't been clear, which is the reason we need several rounds of tests. In this scenario, we need to save the current state, recover the production environment, wait for the next test window, and then use the saved state as the input of the next round to continue testing, until the system is finally stable.

3.3 Environment recovery

After the simulation tests are finished, we will perform following steps to recover the environment:

- (1) Stop the simulation engine on the CDN node.
- (2) After all the synchronous requests have been stopped, we still need to monitor the system to make sure asynchronous requests, such as the messaging system, have been processed.
- (3) Take the applications offline from the expanded servers, and leave only the applications that are running on the servers used in the daily environment.
- (4) Monitor the performance counters of the applications in the daily environment.
- (5) Save the simulation test log, and delete the simulation data from the shadow database and the shadow cache.

4 EVALUATION OF ALL-SPARK

4.1 Capacity planning and error rate evaluation

We defined two counters to evaluate the test result as follows:

- Error rate: The definition is the same as that given in Section 3.2.2.
- Volume difference: It is defined as follows, in percentage: $(\text{capacity planning} - \text{minimum capacity}) / \text{minimum capacity}$. minimum capacity is the minimum number of machines needed in boundary condition we discussed in Section 3.2.

In 11, the x-axis is the volume difference, and the y-axis is the error rate. And the Zero point means minimum capacity with zero error rate. From the perspective of the business requirements, 2% is an acceptable threshold of error rate, and the closer it is to zero, the better. The triangle points represent the output (volume difference and error rate) after running the simulation test in the environment without any bottleneck fixing and fine-tuning based on the capacity calculated from Section 2.1.2.

And the circle points represent the output (volume difference and error rate) of capacity planning result from All-Spark simulation test.

If the volume difference is negative, this means that the predicted number of machines is less than the minimum, then this prediction

Figure 11: Capacity evaluation

is almost certainly to be an unsuccessful one even when the number is very close to zero, as it takes minutes to scale up the machines needed and finally leads to a bad user experience. Therefore, it is important to reserve a buffer to ensure that the predicted capacity is larger than the minimum capacity.

The following conclusions were drawn:

- (1) Almost all the theoretical prediction outputs (triangles) did not meet the error rate target (2%), while all the simulation test results (circles) did.
- (2) The four triangles on the left side of the y-axis denote a relatively low error rate and low volume difference, as they were generated early in 2013 when the big promotion volume was not very high. Therefore, the system was not under a high pressure, and not many bottlenecks occurred. The reason that the predicted number was less than the real requirement was that the bottleneck was not predicted. When a problem occurred on the day of a big promotion later, we had to use a work-around by adding machines, which led to a negative volume difference.
- (3) The other eight theoretical prediction outputs on the right side of the y-axis had a high error rate, and there was a large gap in the volume difference. They were generated when the big promotion had a large volume than that on a normal day; therefore, the larger the number of bottlenecks on the promotion day, the higher was the error rate.
- (4) The simulation test result (circle points) showed all stable results good predictions with the error rate in control. Among all these outputs, circle-1 was the best, as it had the smallest volume difference and the almost zero error rate.

4.2 Bottleneck detection

As mentioned in Section 3.2, once a problem is identified to a certain application, the application owner needs to investigate and solve the problem quickly. Below are some typical issues that we summarized from the previous years' analysis.

4.2.1 Footprintless node.

- **Resource Contention issue:** This is the example we used in Section 1.2. Herein, the two important and high-resource-consuming applications had their virtual machines deployed on the same physical machine and caused the contention. Such an issue is extremely difficult to detect by theoretical predictions before All-Spark was developed. The most severe resource contention found by All-Spark was when all the memory on the physical machine was consumed by 2 applications deployed on it. In an unexpected situation, both applications required a large amount of memory at the same time, resulting in very high memory usage in the physical machine. These two applications had been hung for a very long time, until a process of one application was killed by the operating system "OOM killer".
- **Configuration error:** Any manual configuration setting has the possibility of mis-configuration. One typical error found by All-Spark was the network load balance(NLB) configuration problem, which caused an unbalanced work-load inside a cluster and finally caused an avalanche to destroy all the machines in the cluster. Another error found by All-Spark was incorrect router configuration, which caused a number of cross-area service calls and finally used up the network bandwidth. These configuration issues are not detectable by theoretical predictions.
- **Resource consumption:** Besides the common resources such as CPU, memory, and network bandwidth, All-Spark helps resource bottlenecks. It once found massive amounts of logs generated endlessly after an application error and filled up the disc.

4.2.2 Footprint node.

With respect to the footprint node, the situation is more complicated, as the data itself could be a bottleneck, such as a hotspot. In order to make the simulation as close to the real environment as possible, we usually make a full copy for the basic business factors to create the shadow database and then use All-Spark to reveal the issues that cannot be found by code review.

- **Read and write racing concurrency issue:** This is a typical scenario when one node provides both read and write services. To ensure the data consistency, read and write are usually exclusive. Then, if there are a large number of read and write operations in a random order, the node performance may drop severely. If this issue is discovered, usually, we need to modify the code logic and ensure the correct consequences for the read and write operations.
- **Hotspot issue:** The cache may have some "hot" keys that are accessed concurrently by a large number of requests. When the cache expires that key, subsequent requests after the expiry may overload the backend database (until the cache reloads the data).
- **Cold data issue:** To increase the performance of the data query, the idea of separating cold data and hot data is widely used and a common approach is to store the data being less queried into a file. Further, when a volume spike occurs with a query for the cold data (data stored on file), it will trigger I/O operations and cause a bottleneck for the disc I/O.

Such an issue can be identified by conducting a pulse test. To solve the problem, we developed a warm-up system to preload the potential hot data into the cache, and to thus convert the cold data into hot data before the start of a big promotion.

4.3 Capacity fine-tuning

It is possible that after all the known issues in the bottleneck are checked and excluded, the system load is still high and RT is still large. This usually indicates an insufficient capacity issue, caused by the extra cost incurred while connecting two modules or even two single operations. For example, reading a DB record could cost 1 ms and writing a DB record could cost 5 ms; however, if these two actions take place simultaneously, the time needed may not be 5 ms but actually 8 ms. the additional time is consumed by the read/write lock operations required when using the same database.

Therefore, after all bottlenecks were identified, increasing the server count would be a simple way to address the capacity problem.

5 ALL-SPARK IN OTHER SCENARIOS

We developed All-Spark for simulation testing before a big promotion. Furthermore, All-Spark can be extended to a number of other scenarios.

5.1 Maintain performance baseline for a single node

From time to time, we need to conduct a stress test to evaluate the maximum workload from a single node, then use this data to perform capacity planning as the performance baseline (to calculate the minimum number of servers needed). Such a workload evaluation can also be performed by the All-Spark system.

The performance of a node in assessment might be affected by the nodes that it depends on. To reduce this impact on the measurement, we can simply return mock results from these nodes to the node in assessment. What we need to do is to change the switch on the middleware before server node, to set "join simulation = false", and leave the middleware of testing server unchanged. Then in Figure 6, the excluded node (actually is the middleware) will directly return mock result when dealing with "join simulation" check, so will not join stress test in this scenario.

In 11, we want to test the maximum workload for node-B. So we can create simulation requests (with join simulation = True), and set the switch off on middleware of node C1 and C2. Then when node B sends request to C1, C2, mock result will be returned immediately. So the workload of node B will be purely affected by its own performance, and will not be affected by performances of C1,C2.

Therefore, only node-A and node-B will handle simulation request and nodes-C1 and C2 will not be affected. Thus we can get the maximum workload of node-B.

By using this method, we can conduct 5,000 performance baseline tests on 200 applications every year.

Figure 12: Maintain performance baseline for a single node

Figure 13: Integration test for a new component

5.2 Integration test for a new component

Usually, when the coding work for a new component is completed, we need to perform an off-line test. We need to mock the applications that it needs to communicate with in order to test all its functions. With All-Spark, we can save the work related to the mock applications and perform the online test directly with the simulation data.

In Figure 13, node-B contains the newly developed function, and we want to test its integration with nodes-A, C, and D. Therefore, we can link node-B into the production environment, create a simulation request, and set "join simulation = True" on the middleware before all of these four nodes. Then, the simulation request will be passed and calculated on these four nodes, and stored in the shadow area. At the same time, real requests can still be handled by nodes-A, C, and D, and stored in the real data area. Therefore, the test causes no pollution to the production environment. Currently, in our company, all the new applications need to pass the integration test through All-Spark before going online.

Figure 14: Test for gray users

5.3 Test for gray users

With respect to the existing function with a logical change, we usually launch an internal test to beta users (or called gray users) before opening to all the users.

The traditional solution is to build the test environment with the necessary server nodes (similar to the off-line test discussed in Section 5.2) and invite the gray users to take the test. Although an additional cost is incurred in the test environment setup, it can ensure that the production environment is not affected and the debugging or fine-tuning is easy to perform. An online test can also be performed in addition to the off-line test. However, this requires additional work to modify the processing logic in the caller node (to process differently on the basis of the user list). Moreover, this can easily pollute the production environment if any unexpected result is generated by the updated function and further affects the downstream node. With All-Spark, we can perform the online test as follows.

The only change is in the middleware, to set the blacklist/whitelist for users. That is, a simulation request from a gray user is sent to the shadow server with the new logic, and the related data is saved in the shadow database. A request from a normal user is sent to the normal server with the old logic, and the data is saved in the real data area.

6 CONCLUSION

All-Spark was developed in response to a strong business need to accurately predict the capacity during a big promotion, as the traditional capacity planning methods have been proven to not be applicable in the extremely complicated environment with a high incoming request volume. From a postmortem analysis, we inferred that some bottlenecks exposed only during the big promotion could not be revealed by traditional theoretical prediction. This motivated us to focus on simulation testing.

With the development of the simulation flag and the middleware usage, we solved the key problem of data quarantine, which enabled us to implement online simulations (simulations in the production environment). In the hundreds of tests conducted in the past five years, All-Spark has always generated a capacity evaluation with a

small volume difference (<5%) from the real capacity, which was considerably better than a pure prediction result (>15%).

Moreover, All-Spark contributed considerably to bottleneck detection. Every year around 500 new bottlenecks were detected by All-Spark (because of the integration with new business applications) before a big promotion. Some of them could not be resolved by adding more machines. All-Spark also had a lower error rate (<2%) than that of a pure prediction result (>10%). Besides the concept and the output of the capacity evaluation, All-Spark could be extended to other online test scenarios, because of its features of easy configuration and zero pollution of the business data.

7 FUTURE WORK

We will continue to work on following items in future:

- *Simulation standardization*: We will continue to focus on central management for different middleware and develop a standard for simulation data detection and redirection, in order to generate an easy-to-use solution in any general large traffic environment.
- *Bottleneck removal in early stage*: On the basis of the rich experience of detecting and solving bottlenecks under high workload, accumulated over the years, we are working on performance bottleneck categorization and providing feedback to the infrastructure and application teams to improve the code logic and finally avoid certain bottlenecks from the very beginning.
- Besides the three extended scenarios discussed in Section 5, we plan to find more applicable scenarios that can utilize the advantages of this solution, such as easy configuration on the middleware, online processing, and zero pollution.

ACKNOWLEDGMENTS

We thank our shepherd, Atul Adya, and the anonymous reviewers for their valuable feedback.

And special thanks to Claude, without your encouragement and support, we couldn't finish it.

REFERENCES

- [1] [n. d.]. *ACCS*. <https://blog.csdn.net/ShareUs/article/details/74209139>.
- [2] [n. d.]. *CND*. https://en.wikipedia.org/wiki/Content_delivery_network.
- [3] [n. d.]. *docker*. <https://github.com/docker>.
- [4] [n. d.]. *dubbo*. <https://dubbo.incubator.apache.org/>.
- [5] [n. d.]. *Load Testing*. https://en.wikipedia.org/wiki/Load_testing.
- [6] [n. d.]. *Loader: Simple cloud-based load testing*. <http://loader.io>.
- [7] [n. d.]. *Pouch*. <https://github.com/alibaba/pouch>.
- [8] [n. d.]. *RocketMQ*. <https://rocketmq.apache.org/>.
- [9] [n. d.]. *tengine*. <https://github.com/alibaba/tengine>.
- [10] Per Nikolaj D Bukh. 1992. The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling.
- [11] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N Calheiros. 2009. Modeling and simulation of scalable Cloud computing environments and the CloudSim toolkit: Challenges and opportunities. In *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*. IEEE, 1–11.
- [12] Daniel Gmach, Jerry Rola, Ludmila Cherkasova, and Alfons Kemper. 2009. Resource pool management: Reactive versus proactive or let's be friends. *Computer Networks* 53, 17 (2009), 2905–2922.
- [13] Laura R Moore, Kathryn Bean, and Tariq Ellahi. 2013. Transforming reactive auto-scaling into proactive auto-scaling. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*. ACM, 7–12.
- [14] Dushyanth Narayanan, Eno Thereska, and Anastasia Ailamaki. 2005. Continuous resource monitoring for self-predicting DBMS. In *null*. IEEE, 239–248.

- [15] Christopher Stewart and Kai Shen. 2005. Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 71–84.
- [16] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. 2016. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services.. In *OSDI* 635–651.
- [17] X Zhu, D Young, BJ Watson, Z Wang, J Rolia, S Singhal, B McKee, C Hyser, D Gmach, R Gardner, et al. 2008. Integrated capacity and workload management for the next generation data center. In *ICAC08: Proceedings of the 5th International Conference on Autonomic Computing*.