## Requirement

…

## Design consideration

### Requirement analysis

By definition, the Fibonacci number are the numbers that every number after the first two number are the sum of the two preceding ones, shown as following:

* 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Here is the formula:

* F(n) = F(n-1) + F(n-2), where n is integer and n > 1.

With seed values:

* F(0) = 0, F(1) = 1

It is easy to come up a recursive algorithm shown as below according to the formula. However, this algorithm is problematic because a relatively large number, like 1000, would blow up the process stack.

def Fibonacci(n):

if (n <= 1):

return n

return Fibonacci(n-1) + Fibonacci(n-2)

Here is an iterative algorithm to generate Fibonacci sequence number:

def Fibonacci(n):

a, b = 0,1

yield a

yield b

i = 1

while i <= n:

a, b = b, a + b

i = i + 1

yield b

Although this iterative algorithm’s space complexity is O(1), it’s time complexity is O(n). That is, the time to calculate Fibonacci numbers increases linearly with the parameter n.

Actually, there is a quicker method proposed by the [Prof. Edsgar W Dijkstra](http://www.cs.utexas.edu/users/EWD/) around 1978. With F(0)=0 and F(1)=1, we have:

F(2n-1) = F(n-1)2 + F(n)2  
F(2n) = ( 2 F(n-1) + F(n) ) F(n)

Although the reasoning procedure is not difficult, we ignore it here as it is not our focus. Both the time complexity and space complexity of this method is O(logn) because, essentially, it is a binary tree search algorithm.

So, due to Fibonacci number’s property, the key point to improve the system’s performance is to figure out an efficient and reliable way to reuse/cache the Fibonacci numbers that has been calculated.

### REST Service API

**Title :**

Return a sequence of Fibonacci number

**URL :**

/api/Fibonacci?start=:start&end=:end

**Method :** GET

**URL Params :**

Required:

start=[integer], the start index of Fibonacci sequence

end=[integer], the end index of Fibonacci sequence

**Response Codes:**

Success (200 OK)

Bad Request (403)

**Success Response Codes:**

**Error Response Codes:**

### Architecture



Diagram 1. Overall architecture

The overall architecture is shown in diagram 1.

#### Strategy to calculate and cache Fibonacci number

Here, the responsibility of Fibonacci calculation and Fibonacci request servicing are taken by Spark cluster and WSGI cluster respectively for following reasons:

* Release WSGI servicers’ burden to improve serving latency
* Leverage modern computation framework to compute Fibonacci number in parallel
* Avoid duplicated Fibonacci number computation greatly

According to following formulas mentioned above:

F(2n-1) = F(n-1)2 + F(n)2

F(2n) = ( 2 F(n-1) + F(n) ) F(n)

As illustrated in Diagram 2, Fibonacci numbers calculation can be parallelized in following condition:

Roughly speaking, if (i-1) layer’s Fibonacci numbers have been calculated, the computation of (i) layer’s Fibonacci numbers can be parallelized. With the increase of layer number, this kind of parallelism becomes more effective because the number of Fibonacci number doubles every layer. Theoretically, the time of each layer’s Fibonacci number computation could be constant if there is enough computation resources. We ignore the computation details as each layer’s Fibonacci number computation can be treated as typical Map Reduce program. By the way, for the sake of efficiency, the computation result should be loaded/inserted into the distributed key-value store in batch instead of one by one.

As for the distributed key-value store, popular key-value nosql, like HBase, could be a good candidate as they already take good care of system’s scalability, availability, performance and etc…



Diagram 2. Fibonacci sequence number organized

#### Strategy to serve Fibonacci request

After receiving requests from different clients, Nginx servers work as load balancer to pass the request to WSGI servers. For each WSGI server, it executes following pseudo code:

// Fibonacci(n) is the number that client requests and

// Fibonacci(m) is the largest number cached in Key-Value store, which means that all Fibonacci(k) [k<m] have also been cached. System Management Service will notify each WSGI servers to update this value once a batch of Fibonacci number has been loaded into the distributed key-value store.

if (m >= n):

Retrieve Fibonacci(n) from cache directly and return

else:

Layers = log(n – m) // estimate the cost of computation

If Layers <= LAYER\_CONST: # LAYER\_CONST is a configurable parameter

Calculate Fibonacci(n) according to following formulas:

F(2n-1) = F(n-1)2 + F(n)2

F(2n) = ( 2 F(n-1) + F(n) ) F(n)

Else:

Just return and info client that the Fibonacci number is too large for the system to calculate for now

In this way, WSGI servers rely on distributed cache heavily to satisfy client request very fast and just invoke light weight computation if necessary. And they also possesses some kind of intelligence to protect itself from being ruined by denying really big Fibonacci number request that is out of its current capability.

#### Strategy to trigger Fibonacci number computation

Are the clients happy with this Fibonacci number service?

Does the distributed key-value store cache enough Fibonacci number to service our clients?

Should larger Fibonacci number be calculated and cached in advance?

To answer these questions, we have to collect data from WSGI servers. One typical way is to ask each WSGI server to write a log in its local file system for each request it serves, no matter it is successful or not. And some kind of log collection/analysis system, like ELK, could be leveraged to aggregate and analyze the log. In this way, our system management servers can easily retrieve different kind of service metrics, like 99th percentile latency. Then, it can easily make a decision if it should trigger Fibonacci number computation according to SLA committed to our clients.

## Implementation

Due to the time constraints, I just implement a simple demo

### Deployment

su - root

ssh-keygen -t dsa

ssh-copy-id root@hostname

### Monitoring