# ClientAsyncPrefetchScanner improvement

## Concept

There’s two threads coordinate with each other to achieve async prefetch feature in the old way.

a) IO thread: the thread that pull and waiting data from HBase server, in other words, it’s our dedicated daemon thread that do the work.

b) User thread: the thread that get data from client data cache, the data in the client data cache is pulled by the IO thread.

In the new way, there’s three parts coordinates with each other to achieve async prefetch feature.

a) User thread: the thread that get data from client data cache.

b) Prefetch task: the runnable task that pull and waiting data from HBase server.

c) Prefetch thread pool: the thread pool to execute the task.

## The shortcomings of the original way

a) The shortcomings of Lock.

We use a Lock to coordinate two threads in the original way, using Lock means two threads can't work on the same time, when IO thread is pulling data from server and cache size is lower than limit (in other words function prefetchCondition() return true), user thread will block even cache is not empty. This reduces the performance.

b) The shortcomings of dedicated thread.

Scan api has a high frequency of use, if a beginner forgot, wrote a bug(for example, call close method outside the finally block), or just didn't know close the scanner after use, and the results weren't full scanned, the dedicated daemon thread will not exit which leave a memory leak risk.

## The improvements of the new design

d) Using LockSupport to avoid user thread block while IO thread pulls from server.

LockSupport support coordinate thread without using Lock, means IO thread and user thread can work together on the same time, which improve the performance.

e) Using thread pool to avoid thread waste and memory leak risk.

Using thread pool has two advantage.

(1) Solved memory leak risk above.

(2) Save memory, in jvm, one thread allocates 2M memory in default.

## New Design

The big picture is simple, user thread get data from client data cache and return data if success. If cache is empty, acquire pull lock(the AtomicBoolean fetching attribute), if acquired, pull from HBase server by itself(in case of

prefetch execute pool is busy and prefetch task delayed a lot), or failed, parked until prefetch task pull data from HBase server finish and unpark user thread, and then user thread recheck data cache, meanwhile, if cache size matches prefetch condition, user thread will add a task to prefetch thread pool to pull from HBase server asynchronously.

The detail is a little complicated, we must deal with a little more situation.

The prefetch task maybe delayed in some cases, this led to the user thread take a long time on block status to wait for notify from prefetch task. how do we deal with this?

Because of prefetch task maybe delayed a lot in some cases, for example, the thread pool just has one thread, named T, and T is executing a long time task, the prefetch task wouldn't be execute until the long time task finish, the user thread will block and wait until prefetch task notified it, to avoid this situation, we let user thread pull data from HBase server by itself if prefetch task delayed to avoid wasting time on block and speed up scan.

The loadCache method can’t be execute concurrently, so we must avoid user Thread and thread in execute pool execute loadCache method concurrently, use a lock to do this is fine but lock will lower the performance, what we need here is, if user thread acquire the rights to execute loadCache method, we execute it, if user thread failed, user thread just park and wait prefetch task execute loadCache method for it, so use AtomicBoolean is a better way to do this.

When does USER Thread execute loadCache method?

a) Prefetch Lock acquired

b) Cache is empty, it could be a prefetch task delay that caused it.

c) Scan is not closed.

Step one:

If cache is empty, first, user thread checks if the scanner is closed, if scanner is closed, return null.

Step two:

user thread checks if user thread can obtain prefetch lock (the AtomicBoolean attribute), if obtain lock success, recheck if cache is empty and if scanner is close, if met, execute loadCache method, if not, recheck from step one.

if obtain lock failed, just park to wait prefetch task to unpark itself.

When does user thread park?

If cache is empty, scan is not close, but user thread failed to acquire prefetch lock, that means prefetch task is pulling data from HBase server, user Thread will park and waiting prefetch task to unpark it.

When does prefetch task execute loadCache method?

a) Lock acquired

b) Scan is not closed.

c) Prefetch condition is met.

Step one:

acquire lock.

Step two:

if step one success, check if scan is not closed and if prefetch condition is met, if all two condition met, execute locdCache method, if not, just exit.

When does prefetch task to unpark user thread?

a) When loadCache method executed, that means prefetch task changes at least one things of data cache or close attribute, unpark user thread so user thread can get data from cache or just return null when scanner is close.

b) When loadCache throws an exception, data cache and close attribute may not be changed, we need to unpark user thread again after AtomicBoolean set to false, so we can let user thread have the chance to execute loadCache itself and run again if user thread parked on that time.

When to add prefetch task to execute pool?

a) Prefetch condition is met.

b) There is no prefetch task in thread pool.

we maintain one prefetch task in execute pool because if execute pool is busy, add more prefetch task is useless, here we use a bool type named added to indicate if execute pool have a prefetch task inside, set added to true when added a prefetch task to execute pool and set added to false when task is finished, before add prefetch task to execute pool, we must check added attribute to determine if there is a prefetch task already.

How we deal with multi scan threads?

For LockSupport api, unpark method need a Thread parameter, which is user thread that may park when met the park condition, we save this parked thread in one AtomicReference attribute, when another thread, named T, execute next() method, it will park when met the park condition, but prefetch task can't unpark it because it can't acquire T's reference, so according to LockSupport's annotation, T will unpark in a random time and rerun, so to avoid this, we check the scan thread to see if it is the original

one, if not, throw a DifferentScanThreadException to avoid thread block for a long time.

## Some other improvement

f) If we created a lot of ResultScanner from one HTable, the default thread pool which is created by HTable may slow down scan, so we let user to decide whether or not use a custom execute pool instead of default one to speed up.