EAI Endorsed Transactions
on Scalable Information Systems
Research Article EAI.EU

Smart Data Prefetching Using KNN to Improve Hadoop Performance

R. Ghazali,[1,2*] and Douglas G. Down [2]

[1] Department of Computer Engineering, North Tehran Branch, Islamic Azad University, Tehran, Iran
[2] Department of Computing and Software, McMaster University, 1280 Main St W, Hamilton, ON, Canada

Abstract

Hadoop is an open-source framework that enables the parallel processing of large data sets across a cluster of machines. It faces several challenges that can lead to poor performance, such as I/O operations, network data transmission, and high data access time. In recent years, researchers have explored prefetching techniques to reduce the data access time as a potential solution to these problems. Nevertheless, several issues must be considered to optimize the prefetching mechanism. These include launching the prefetch at an appropriate time to avoid conflicts with other operations and minimize waiting time, determining the amount of prefetched data to avoid overload and underload, and placing the prefetched data in locations that can be accessed efficiently when required. In this paper, we propose a smart prefetch mechanism that consists of three phases designed to address these issues. First, we enhance the task progress rate to calculate the optimal time for triggering prefetch operations. Next, we utilize K-Nearest Neighbor clustering to identify which data blocks should be prefetched in each round, employing the data locality feature to determine the placement of prefetched data. Our experimental results demonstrate that our proposed smart prefetch mechanism improves job execution time by an average of 28.33% by increasing the rate of local tasks.

Keywords: Hadoop performance, Smart prefetch technique, K-Nearest Neighbor clustering, MapReduce, Machine Learning, Cache replacement

Received on 28 August 2024, accepted on 01 November 2024, published on 17 April 2025

doi: 10.4108/eetsis.9110

1. Introduction

Hadoop [1] enables the storage and analysis of large datasets by incorporating two main components. The first, MapReduce [2], is a parallel programming model for processing a large amount of data through a cluster of machines in a distributed environment. The second component is the Hadoop Distributed File System (HDFS) [3] to holds a large volume of data. Although Hadoop can bring significant benefits it suffers from problems such as access latency in reading data from HDFS and increased input data transmission time from a remote node to a processing node. These issues can lead to prolonged runtimes. Prefetching has been proposed as an efficient way to accelerate execution by mitigating these delays. We could classify a prefetching strategy into two groups based on the source of fetching: HDFS prefetching to improve data access time from HDFS and node prefetching to reduce data transmission time through the network. For this purpose, a prefetching thread is created to fetch data into the cache before they are requested. HDFS prefetch faces some challenges. Due to the master/slave architecture of HDFS, prefetching is a two-step process where the steps must be made compatible: Metadata prefetching, and Data block prefetching. Metadata prefetching from the NameNode mitigates access latency for metadata and reduces NameNode overhead. Data block prefetching from worker nodes reduces

*Corresponding author. Email: Ghazalir@mcmaster.ca

I/O overhead. In both steps, we identify some issues for efficient prefetch:

- Prefetching time: At what point in time is it appropriate for the prefetch process to be triggered? If data blocks are cached too long before they are requested they will have little chance of being accessed. Due to limited cache space, there is a high probability that these data blocks will have been evicted from the cache at the time that they are needed. Therefore, we should determine a suitable prefetch time in terms of system parameters such as node processing capacity, task processing time, and on-demand data access time.
- Prefetched data: Which data should be prefetched in each round? We should be aware of requested data in each step to prefetch on-demand data just before they are processed. If we prefetch the wrong data, on-demand data will not be accessed from the cache leading to high data access time.
- Location of prefetched data: Where is a suitable location to store prefetched data? Data locality is an important factor that has a significant impact on decreasing job execution time by reducing data transmission time. In other words, we should take into account the shortest distance between the processing node and the location of the prefetched data. Therefore, the cache of the processing node is the ideal place. If the cache capacity of the processing node is full, caches of neighboring nodes can be considered.
- Prefetched data volume: How much data should be prefetched in each step? By considering cache size, data block size, and node processing capacity, we would like to determine a suitable volume of data to prefetch. If these data volumes are too large the cache not only contains data items that have not been used recently but cache pollution may also arise. On the other hand, the frequency of prefetching will increase if prefetched data volumes are too low. This leads to poor performance as a result of increased overhead.

There are different lines of research that consider each of these issues individually in the prefetch mechanism, but we are not aware of any work that considers all of them together. In this paper, we present a smart prefetch strategy using KNN (K Nearest Neighbors) clustering [4] to determine which data (including volume) should be prefetched in each step. Moreover, we determine a suitable time to launch the prefetch mechanism by considering node processing capacity and task processing time. Finally, data locality considerations determine where prefetched data should be placed. Our contributions in this paper are:

- We provide a brief overview of existing prefetch mechanisms and discuss their advantages and disadvantages.
- We propose a novel prefetch mechanism that calculates the prefetch time and uses the KNN cluster algorithm to determine which data should be prefetched and where it should be placed.
- We carry out experiments to investigate the impact of our proposed prefetch mechanism on Hadoop performance.

The rest of the paper is organized as follows: We discuss existing prefetch strategies for Hadoop and compare their advantages and disadvantages in Section 2. We then describe the proposed framework and present our KNN-based prefetch algorithm in Section 3. The performance of the proposed prefetch method is evaluated via different experiments in Section 4. Finally, Section 5 contains conclusions and suggestions for future work.

2. Related work

There are various mechanisms to enhance Hadoop performance in different aspects like improving data locality rate, speculative execution, and fair resource distribution [5]. Considering data locality ensures data is stored close to where it will be processed; avoiding data transmission has a positive effect on data access time [6] [7]. In-memory caching approaches enable frequently accessed data to be cached. One important aspect of in-memory caching is the prefetching mechanism, which is the main topic of this paper.

Since BigData applications contain massive numbers of data blocks, there is no guarantee that all tasks can obtain their input data blocks from the cache. Due to the large amount of data, data blocks may be evicted from the cache before they are required. In [8] Just Enough Cache (JeCache) was proposed as a solution for this problem using a just-in-time data block prefetching mechanism. This mechanism monitors data block access and calculates average data processing time to determine the minimal number of data blocks that should be kept in the cache. JeCache consists of two parts: 1) Prefetch information generation uses job history logs to determine which data blocks should be cached initially and a sequence of data blocks that need to be prefetched when a job is running. 2) The prefetch controller monitors data block access in each worker node, evicting data blocks from the cache when their processing is finished. This mechanism can reduce cache resource demand and improve execution times. However, it only considers read caching.

Vinutha et al. [9] introduced a solution to decrease data transmission time between a remote node and a processing node in a heterogeneous cluster. For this purpose, a prefetch thread is created to fetch requested input data in advance from a remote node to the buffer of the processing node, which is used as temporary storage. This results in a positive impact on job execution time by overlapping data transmission with data processing and increasing the data locality rate when launching a task. Even with this prefetching strategy, a job waits for the first data transmission. In [10] a streaming technique was presented to address this problem. In this method, data transfer and data processing are performed simultaneously. The resulting smaller size of the streaming data can reduce transmission waiting time. Kalia et al. [11] proposed speculative prefetching that takes into account node processing capacity to load input data into the processing node. It groups intermediate data via the KNN clustering algorithm using a Euclidean distance measure to improve the

data locality rate for Reduce tasks, leading to enhanced performance. However, it does not consider other features like workload capabilities and worker node throughput. In [12] a two-level correlation-based file prefetching mechanism and dynamic replica selection were introduced to reduce data access latency and avoid overloaded worker nodes via load balancing. In this strategy, four placement patterns are considered to store fetched data.

Table 1 presents a comparison of these prefetching strategies. Each of these strategies concentrates on only one prefetching issue. For instance, JeCache focuses on choosing an appropriate prefetching time and speculative prefetch considers node processing capacity for load balancing. We attempt cover a combination of these issues in our proposed prefetch mechanism.

Table 1. Comparison of prefetching strategies

Technique	Prefetch type	Location to store prefetched data	Advantages	Disadvantages
JeCache [8]	HDFS prefetch	Worker node cache	Uses cache space efficiently	Only considers read cache
Prefetch thread [9]	Node prefetch	Processing node buffer	Reduces transmission time	Waiting time for first data transmission
Streaming technique [10]	Node prefetch	Processing node buffer	Reduces waiting time	Does not load balance
Speculative prefetch [11]	Node prefetch	Worker node cache	Considers node processing capacity	Does not take into account node throughput
Two-level correlation-based file prefetching [12]	HDFS prefetch	Considers four patterns (ND-pattern, CD-pattern, NC-pattern, and CC-pattern)	Dynamic replica selection	Does not consider some features

3. Methodology

In this section, we present the design of our proposed prefetching mechanism by addressing the key issues discussed in the introduction. Firstly, we determine when to launch the prefetching mechanism based on the progress rate of tasks. Secondly, based on the processing capacity of the worker node and the available cache space, we calculate the number of data blocks that can be prefetched in each round, K, using the KNN algorithm. Finally, we determine the location for prefetched data based on the data locality rate to reduce execution time by minimizing data transmission time. A sequence diagram is presented in Figure 1 to clarify the workflow of our proposed prefetching mechanism.

3.1. prefetch time

In theory, the optimal timing for launching prefetching in Hadoop is determined by when the prefetched data can be fully utilized by Map or Reduce tasks, and when it can be made available in the cache before it is needed. This approach can help minimize the impact of data access latencies on Hadoop job performance. One method of determining the best timing is to use the progress rate of Map and Reduce tasks to predict when they will require data. In this case, a threshold value should be set for the progress rate of Map and Reduce tasks. Once the progress rate exceeds the threshold, the prefetching mechanism can be launched to begin fetching data blocks that are likely to be needed soon. By launching prefetching at this time, the data can be made available in the cache before it is needed, reducing data access latencies. To avoid interfering with other tasks or causing excessive network traffic, it is advisable to trigger prefetching when the system is relatively idle, such as during periods of low job activity or off-peak hours. Ultimately, the optimal timing for prefetching in Hadoop depends on various factors, including the job's characteristics, worker node processing capacity, and network resource availability. Monitoring system performance and adjusting the prefetching strategy as needed is critical to achieving favorable results.

To determine the best time to launch prefetching, we calculate the progress rate of tasks and determine a suitable threshold value. To calculate this threshold, we first introduce some notation: a job consists of T tasks (either Map tasks or Reduce tasks), with each task processing N <key, value> pairs, the number of processed pairs is M, and the task has completed L stages (for Reduce tasks, there are three stages: copy data phase, sort phase, and reduce phase). The progress rate of the ith task, PS_i, is estimated based on the percentage of the task's <key, value> pairs that have been processed, as shown in Eq. 1 [14]. The average progress rate of a job, PS_{avg}, is then calculated using Eq. 2. Furthermore, the progress rate of a task T can be computed based on how many <key, value> pairs are processed per second, given the task has run for T_r seconds, as shown in Eq. 3 [15]. By setting a suitable threshold value for the progress rate of Map and Reduce tasks, the prefetch mechanism can be triggered when the progress rate exceeds the threshold value, indicating that the task will soon require data blocks that can be prefetched.

$$PS_i = M/N \quad \text{for Map tasks}$$
$$PS_i = 1/3(L+M/N) \quad \text{for Reduce tasks} \quad (1)$$

$$PS_{avg} = (1/T) * \sum_{i=1}^{T} PS_I \quad (2)$$

$$PR_i = PS_i/T_r \quad (3)$$

To determine the threshold value for launching the prefetch mechanism, we need to consider the workload and system characteristics. The choice of threshold should balance the risk of triggering prefetching too early (leading to wasted network bandwidth and cache space) and the risk of triggering prefetching too late (resulting in longer processing times due to data access latency).

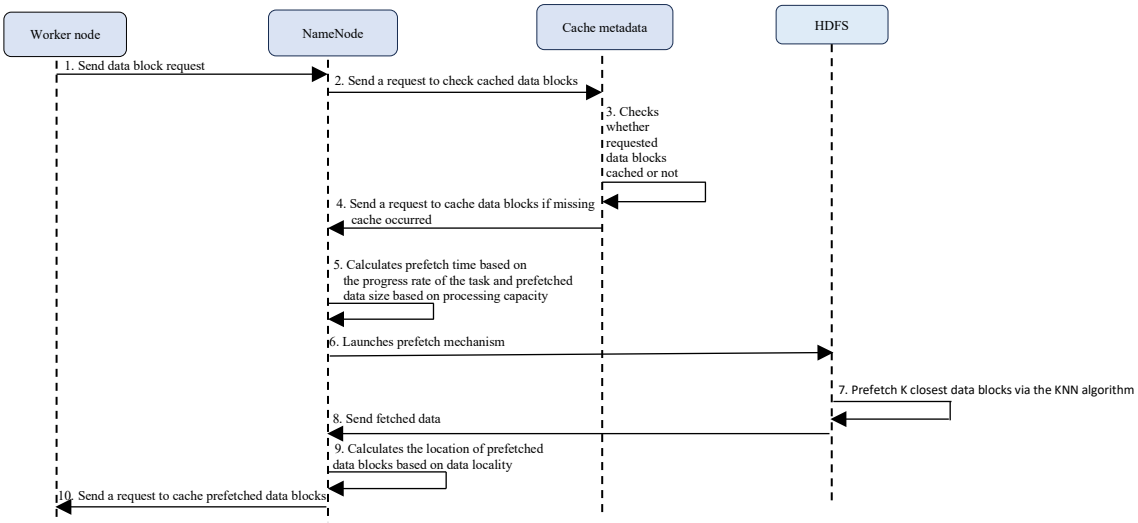

Figure 1. Sequence diagram

In this paper, we obtain this threshold directly, as will be seen in our experimental results (Section 4.3). Another possibility would be to find a parameterized function to determine the threshold. A potential formula is expressed as Eq. 4, where α and β are scaling factors that can be adjusted based on the characteristics of the workload and cluster. To determine α and β, one could consider workload features such as cache affinity and job type (I/O-bound and CPU-bound), while the cluster specifications could include available cache space and worker node processing capacity. To calculate the standard deviation of the progress rate of Map and Reduce tasks across the entire job, we use Eq. 5, where T is the total number of progress rate values. By taking into account these factors, we can determine an appropriate threshold value for launching the prefetch mechanism that minimizes the impact of data access latencies on the performance of Hadoop jobs.

$$\text{Threshold} = \alpha * (PS_{avg} + \beta * PS_{std}) \quad (4)$$

$$PS_{std} = \text{sqrt}\left[\sum_{i=1}^{T} (PS_i - PS_{avg})^2 / (T - 1) \right] \quad (5)$$

3.2. prefetched data size

The volume of data to prefetch is a crucial factor affecting performance. Insufficient prefetched data can result in the system having to retrieve demand data from the disk or network, leading to increased data access time or an increased frequency of prefetch operations. On the other hand, excessive prefetched data can lead to excessive load on a worker node and increased resource contention, negatively affecting performance. The number of data blocks that can be prefetched depends on various factors, such as the processing capacity of the node, job running time, and the degree of parallelism. In a Hadoop heterogeneous environment, where the cluster consists of a set of worker nodes with varying processing capacities and cache sizes, the prefetched data size (number of data blocks) can be calculated using a relation such as Eq. 6. Here, W is the number of worker nodes, Pj denotes the processing capacity of the jth worker node, and Cj represents the available cache space on the jth worker node. The equation takes into account the parallelism degree, denoted by Nc (the number of tasks that can run concurrently), the estimated average processing time of a data block in the jth worker node (ETj), and the data block size (BS). Moreover, we adjust the value of K for specific workloads as some workloads may benefit from a smaller K, while others may perform better with a larger K. For this purpose, the cache affinity of applications (CA) [16] is considered as a coefficient that determines how to utilize the benefit of cached data in each application. For our purposes, we assume that it can be classified into three levels: high, medium, and low cache affinity with values of 0.75, 0.5, and 0.25 respectively.

$$K = \sum_{j=1}^{W} (Pj * ETj * CA * Nc) / BS \quad (6)$$

We should choose an appropriate value for K. If it is too small, the result is under-prefetching which can lead to a lower cache hit ratio and missed prefetch opportunities. On the other hand, if K is too large, the result is over-prefetching via increasing cache pollution. Therefore, the choice of K should manage the trade-off between under-prefetching and over-prefetching. This choice depends on the Hadoop cluster characteristics and features of the workload. We will further explore the choice of K in our numerical experiments.

3.3. KNN clustering

The K Nearest Neighbor Classification (KNN clustering) algorithm [17] is a non-parametric algorithm employed for the identification of the nearest K training instances to a given test data point. Once these K closest training instances are ascertained, the KNN algorithm employs a majority voting mechanism, where the class with the highest frequency among the selected K instances is specified as the classification for the test data. This clustering algorithm contains two key phases:

- **The Training Phase:** During this stage, the algorithm stores the training data point coordinates along with their respective class labels, which serve as identifiers for classifying data points in the subsequent testing phase.
- **The Testing Phase:** This phase entails determining the class labels for new data through a majority vote among the K nearest neighbors of the testing data point, derived from the training dataset using a defined distance metric.

The general procedure of the KNN clustering algorithm can be summarized as follows:

1. Determine the value of K.
2. Preparation of the training dataset, which involves storing data point coordinates and their associated class labels.
3. Loading the data point from the testing dataset.
4. Execution of a majority vote among the K nearest neighbors of the testing data point from the training dataset, based on the prescribed distance metric.
5. Assignment of the class label from the majority vote winner to the new data point in the testing dataset.
6. Repeat these steps until all data points in the testing phase are accurately classified.

Algorithm1: Mapper function

```
Function Mapper()
1-tr =Load training dataset in the form of
<key, value> pairs
// key is the unique identifier for a data block and the value
includes the features and the class label
2- ts=Load test dataset
3- Class_tr = readClassId(value)
4- For i=1 to n   // Iterate over each test instance
5- For j=1 to m// number of training instances
6-distance=distance+ (tsi-trj) ^2
7-distij =SQRT (distance) //Calculates Euclidean distance
8-Return(<i, object <trj , distij, class_tr>>)
9-End for
10-End for
```

Output: <key1, value1> where key1 is the unique identifier for a data block and value1 encompasses information about the distance between this data block and others, along with their associated classes, presented as <DB_i, <DB_j, $dist_{ij}$, class A>>.

Algorithm2: Reducer function

Input: <key1, <List value1's> distances> where key1 is the data block id
of the test instance, value1is an object which contains:
< DB_i, <DB_j, $dist_{ij}$, class A >, distances: List of all distances, and K value

```
Function Reducer(K)
1- Sort_ascending(distances)
2- new LinkedList K_distances
3- new LinkedList Classes
4- For i=1 to K
5- K_distances.add(distances.get(i))
6- End for
7- For all dist ∈ K_distances
8- If dist. getclass()∉ classes
9- classes.add(dist.get class())
10- End if
11- End for
12- For all class_id ∈ classes
13- If dist < min then
14- min = dist
15-decided_class_id = class_id
16-End if
17- End for
18- key2 = key1
19- value2 = decided_class_id
20-Reteun(key2,value2)
```

In the context of our proposed approach, we utilize the KNN MapReduce programming model [18] [19] to cluster data blocks within the Hadoop Distributed File System (HDFS), based on Euclidean distance. In addition to the required data block, we also prefetch the K nearest data blocks as they are more likely to be needed soon. In this framework, the Mapper function is responsible for computing distances between the testing instance to be processed and the training instances contained within the received training data split. Consequently, the output of the Map task is structured as a <key, value> pair, where the key designates a data block to be classified, and the value encompasses information about the distance between this data block and others, along with their associated classes, presented as <DB_m, <DB_n, $dist_{mn}$, class A>>. The Mapper function is presented in Algorithm 1. The Reducer function plays a central role in the classification decision for the test instances. Upon receiving inputs from the output of the Map tasks, the Reducer function identifies the K nearest neighbors from the entire set of training instances by determining the K smallest distances within the list, followed by majority voting. Algorithm 2 provides details of the Reducer function.

In scenarios involving substantial data volumes and an extensive feature set, some data points may exhibit similar proximities to instances from different classes. In such instances, the minimal distance from individual instances may not definitively signify that the test instance belongs to a particular class. For instance, suppose that K is set to 5, and the following values are provided to the reducer for the test instance DB_7:

<DB_7, <DB_2, 0.11, A>>
<DB_7, <DB_{10}, 0.12, B>>
<DB_7, <DB_9, 0.15, B>>
<DB_7, <DB_{13}, 0.13, C>>
<DB_7, <DB_{16}, 0.12, A>>

In this scenario, the classical KNN algorithm, due to its majority voting strategy, would suggest that the test instance DB_7 belongs to either class A or class B. However, further examination reveals that class A contains an instance with the closest proximity to the testing instance, hence leading to the conclusion that DB_7 is most appropriately categorized within class A.

3.4. Prefetched data location

Considering data locality to determine the prefetched data location can have a positive impact on decreasing job execution time. Assume that for a set of worker nodes where all replicas of data blocks are located, D_d is the distance between worker node d and the processing node. Additionally, C_d represents the processing capability of node d (a combination of CPU and memory capacity), while L_d represents the workload of worker node d. It is important to choose an appropriate worker node with a low load and distance to the processing node, to reduce data transmission time, and ensure enough processing capacity to balance the load in the cluster, thereby increasing resource utilization. A high-level conceptual formula that incorporates these parameters is given in Eq. 7, where f is a function that takes into account the specific algorithms and policies implemented in Hadoop to calculate a suitable location for prefetching data. It is obvious that data locality and node processing capacity have direct relationships, as Hadoop prioritizes nodes with local copies of the required data that are closer to the nodes where the task is scheduled. Also, nodes with higher processing capacity are preferred for executing tasks to ensure that the computational load can be handled effectively. When a node is under heavy load or already processing a significant amount of data, it is logical to avoid placing additional prefetched data on that node and to distribute the tasks among less loaded nodes, ensuring a balanced workload distribution. In summary, the selected node for the prefetched data, S, is given by:

$$S=\text{Arg Min}\{f((D_d, C_d)/L_d)\} \quad (7)$$

A similar formula is used in the dynamic replica selection algorithm proposed in [9], where identical parameters are taken into account to determine which data blocks should be replicated, thereby mitigating the occurrence of the hot worker node phenomenon. We do emphasize that the function f is an abstract function used to illustrate what should determine the location of prefetched data. As we will see in our experiments, this notion can be used in a more conceptual manner.

3.5. Proposed prefetch algorithm

Algorithm 3: KNN prefetch algorithm

Input: List of heterogenous worker nodes, Input data size (IDS), List of tasks, Data block size (DBS)

1-TDB=IDS/DBS //Calculate the total number of data blocks
2-If task T_i is Map task // Calculate the progress rate
3-PS_i = M/N
4-Else PS_i =1/3*(L+M/N)
5-End if
6-PR_i = PS_i/ET_i
7-TimetoEnd$_i$ = (1 − PS_i)/ PR_i
8-PS_{avg}= (1/T) * $\sum_{i=1}^{T} PS_i$
9- If TimetoEnd$_i$>= Threshold* PS_i then launch prefetch
10. K= $\sum_{j=1}^{W}(P_j * ET_j * CA * N_c)/BS$ //Calculate K value
11-Call Mapper()
12-Call Reducer(K)
13-Look up block metadata to find the location of these K data blocks //Data locality for prefetched data
14-If they are not located in the worker node then
15- Worker nodes=Arg Min{f((D$_d$, C$_d$)/L$_d$)}
16- Send a request to the worker nodes to cache data blocks
17-End If

This section outlines our proposed algorithm for smart prefetching, which we present in detail. The input to our KNN prefetch algorithm includes a list of worker nodes in the cluster along with their processing capabilities, a list of tasks to be performed, and the size of the input data. First, we calculate the number of data blocks based on the Hadoop Distributed File System's (HDFS) data block size. Next, we estimate the time remaining for each task based on its progress, which depends on the task type, using Eq. 2. We then check whether the estimated time remaining for a task has reached the predefined threshold value or not. If it meets the threshold value, the prefetching process is initiated. The number of data blocks that can be prefetched, denoted by K, is determined using Eq. 6. We use K Nearest Neighbor (KNN) clustering to group the data blocks into clusters, with each cluster containing the data blocks that can be prefetched simultaneously during each prefetch phase. We determine the K data blocks that are located in the nearest neighborhood of the required data blocks based on their Euclidean distance as has been described in

Section 4.3. Finally, we place the prefetched data blocks based on the data locality factor, as described by Eq. 7, which reduces the data transmission time through the network and positively impacts the job execution time.
In this algorithm, we assume that workload patterns do not change over time. Otherwise, we need to monitor cache performance continuously and predict the optimal K value based on historical data of access patterns and workload characteristics (using machine learning techniques, for example). Also, it would likely be beneficial to establish a feedback loop to regularly re-evaluate the chosen K value as the system evolves and workload patterns change. Incorporating this increased complexity is something that we are considering for future work.

4. Results and Discussion

In this section, we describe the experimental environment including software and hardware configurations and some Hadoop configuration parameters settings. Our evaluation is divided into two sections: investigating the appropriate threshold value for choosing the best prefetch time and evaluating the impact of smart prefetch on Hadoop performance.

4.1. Experimental setup

For our experiments, we use a cluster consisting of a single NameNode and ten worker nodes located in two racks such that odd-numbered nodes are in rack1 and even-numbered nodes are in rack2.

- *Hardware configuration*: The nodes are connected via a 10 Gigabit Ethernet switch. The experimental environment is a heterogeneous environment with different memory sizes and processing capabilities. Details are presented in Table 2.
- *Software configuration*: We use the Ubuntu14.04 operating system and JDK 1.8, Hadoop version 2.7, and Intel HiBench [20] [21] version 7.1.
- *Hadoop configuration parameters:* The block size of files in HDFS is 128 MB, and data replication is set to 3.
- *MapReduce applications*: We use Intel HiBench as a Hadoop benchmark suite that contains the following applications: 1) WordCount is a CPU-intensive application that returns the number of occurrences of each word in a text file. 2) Sort is a typical I/O-bound application that sorts input data. 3) Grep is a mix of CPU-bound and I/O-bound operations that searches for a substring in a text file. In addition, the cache affinity feature determines how to utilize the benefit of cached data in each application such that it can be classified into three categories based on this feature: low cache affinity (Sort), medium cache affinity (WordCount), and high cache affinity (Grep).
- *Input data:* For carrying out experiments, we have used the default data sizes from the HiBench suite. Sort and WordCount have 60 GB and Grep has 1 TB, respectively, for their input data sizes. The input data are generated by using the RandomTextWriter.

Table 2. Node characteristics

Node type	Hardware configuration	Node processing capability
Master node	Intel Core i7-6700 processor, 2.4 GHz CPU,64 GB memory, 10 data blocks cache capacity, and 1 TB hard disk.	8 Map slots, 5 Reduce slots
Odd worker nodes	Intel core i5-4590 processor, 3.30 GHz CPU, 32 GB memory, 8 data blocks cache capacity, and 500 GB hard disk.	4 Map slots, 2 Reduce slots
Even worker nodes	Intel core i5-4590 processor, 2.9 GHz CPU, 16 GB memory, 6 data blocks cache capacity, and 250 GB hard disk.	2 Map slots, 1 Reduce slot

4.2. Metrics

In our experiments, we consider two key performance metrics:

- *Job execution time:* This plays a vital role in Hadoop performance improvement, and is related to data access time. The data access time decreases significantly if we can access data from the cache instead of the disk, reducing the job execution time.
- *Data locality rate:* This is measured using the total number of tasks run locally in the node where the associated data resides.

4.3. Investigating the threshold value

As we mentioned in the previous section, selecting the appropriate value (between 0 and 1) for the threshold parameter is a crucial issue that depends on various factors. For this purpose, we consider two workloads to investigate the impact of different threshold values on job execution time: WordCount as a CPU-oriented application with medium cache affinity, and Sort as an I/O-oriented application with low cache affinity.

In Figure 2, we can observe that execution time increases when the threshold value is less than 0.6 or more than 0.8. The number of references to HDFS for fetching data blocks rises when the threshold value is too small. Prefetching large numbers of data blocks before requiring them leads to overload. In this case, there is a high probability that some data blocks are evicted from the cache before they are required, leading to a negative impact on job execution time. Also, if the threshold value is set too large (greater than 0.8) a greater number of data blocks are not accessible from the cache when required as a result of needing to wait until the prefetch mechanism is completed. The result is resource underutilization if prefetch is performed too late The choice of threshold should take into account the trade-off between these two issues. For our experiments, a suitable value appears to be 0.7,

independent of the application. It would be of interest to further explore if such insensitivity holds across other applications and/or operational environments.

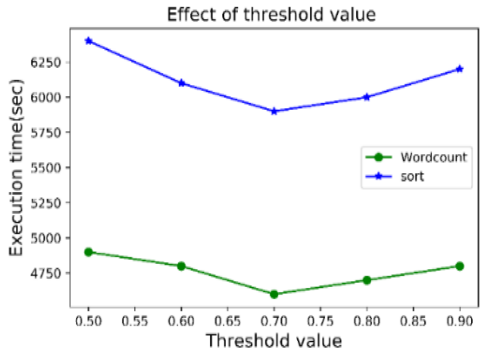

Figure 2. The impact of threshold value on job execution time

4.4. Investigating the impact of K

In this experiment, we consider a cache size of 1024 MB, data block size of 128 MB, and two applications: Grep and Sort as high cache affinity and low cache affinity, respectively. Next, execution time is evaluated based on three values of K including small (2), medium (5), and large (10). Figure 3 illustrates execution time as a function of K for both applications.

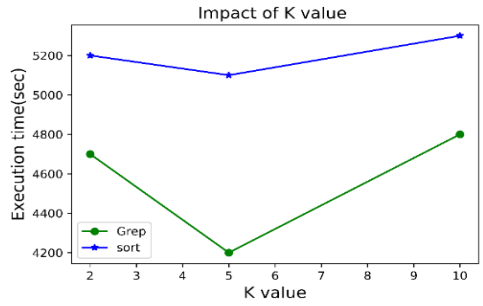

Figure 3. The impact of K on job execution time

The results indicate that both low and high K values result in degraded prefetching behaviors, with low K values causing under-prefetching and high K values causing over-prefetching. Over-prefetching leads to cache pollution in both low-cache and high-cache affinity scenarios, negatively impacting execution time. This is due to the presence of unneeded data blocks in the cache, which reduces the cache hit ratio and increases unnecessary I/O operations to retrieve the required data blocks. Conversely, under-prefetching, associated with small K values, results in frequent cache misses, necessitating data block retrieval from slower storage and thereby increasing execution time. In applications with low cache affinity, the impact of both under-prefetching and over-prefetching on execution time is minimal. However, in high-cache affinity applications, the negative effects on execution time are significant.

4.5. Impact of smart prefetch on Hadoop performance

In this section, we evaluate the performance of smart prefetch based on two metrics: data locality rate and job execution time. For this purpose, we run a combination of the Micro benchmark in Intel HiBench (composed of both I/O-bound jobs and CPU-bound jobs) to compare our algorithm to the following strategies:

- Hadoop original: No caching is utilized.
- Simple prefetch: Data is prefetched into the cache before the actual processing starts with the help of a prefetching thread.
- Speculative prefetch [9]: A recently proposed algorithm that utilizes KNN to cluster intermediate data and as such has similarities to the smart prefetch mechanism.

Performance evaluation based on data locality rate

Figure 4 presents the data locality rate for three applications and compares our smart prefetch algorithm with the three algorithms given above. Smart prefetch has the best local task rate, demonstrating that it improves data locality by locating prefetched data in an intelligent manner. Also, by prefetching high-probability demand data just in time, more tasks have a chance to use cached data. Experimental results show that the locality rate of tasks has improved by 14.2%, 7.8%, and 3.22% in WordCount, 15%, 9.52%, and 2.22% in Sort, 17.07%, 10.34%, and 4.34% in Grep against Hadoop original, Hadoop with simple prefetch, and Speculative prefetch respectively. In addition to showing the possible performance improvement of our algorithm, this also suggests that our method is more suitable for high cache affinity applications and I/O-bound jobs, as they benefit more from cached data.

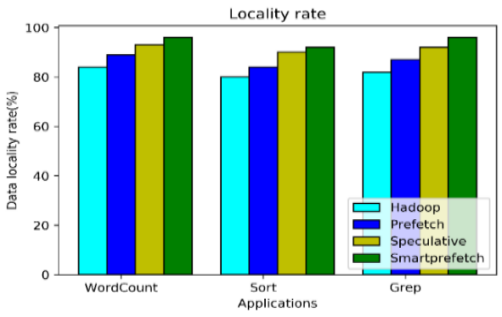

Figure 4. Local tasks rate for different applications

Performance evaluation based on job execution time

In this experiment, we use average job execution time as a performance measure to evaluate our proposed prefetching

strategy. We use the Delay job scheduler that considers the data locality rate in its scheduling policy. We also execute each application 10 times to calculate the average execution time. In Figure 5, we can observe that WordCount, Sort, and Grep finish their jobs 17%, 9.4%, and 16% faster than the default prefetch mechanism and their execution time improves by 30%, 25%, and 30% against Hadoop native and the speed increases by 8%, 3%, and 7.5% compared with Speculative prefetch. Our novel prefetch mechanism improves the data locality rate with a resulting positive impact on execution time due to decreased data transmission time. Also, the cache hit ratio is increased via prefetching the appropriate volume of data, resulting in more tasks having the chance to utilize cached data, improving data access times with a resulting positive impact on execution time.

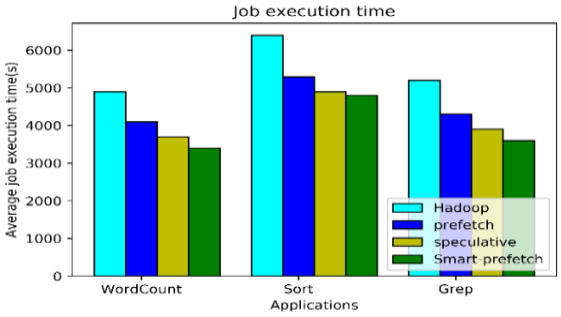

Figure 5. Average job execution time for different applications

5. Conclusion and future work

In this paper, we have presented a smart prefetch mechanism comprised of three phases. Firstly, we determine the optimal launch time for prefetching data blocks from the Hadoop Distributed File System (HDFS) by calculating the progress rate of tasks on worker nodes. This phase's success is limited by the threshold parameter, which must be set based on various environment and workload characteristics. While this paper provides a proof of concept of our prefetching algorithm, further work on how to choose this threshold is required. This could consist of additional guidelines, but ideally one would like to choose/learn this parameter in an automated fashion. Next, we utilize the K-Nearest Neighbor (KNN) clustering algorithm to identify the number of data blocks with a high demand priority that can be fetched in each round. Finally, we evaluate data locality as a metric for placing the prefetched data. Experimental results indicate an average job execution time improvement of approximately 28.33%, 14.33%, and 6.16% when compared to Hadoop original, the default prefetch mechanism, and speculative prefetch respectively. This improvement is attributed to a 15.4%, 9.22%, and 3.25% increase in data locality rate, respectively. In addition to determining the threshold to launch the prefetch mechanism, there are additional parameters that must be tuned. As a step in this direction, we plan to assess the proposed prefetch mechanism's scalability by testing it on a large cluster as well as automatically determining the K value for dynamic workload by utilizing machine learning methods.

References

[1] Apache Hadoop, http://Hadoop. Apache. org/, last accessed 2021/02/15

[2] Khezr, S. N. & Navimipour, N. J. MapReduce and Its Applications, Challenges, and Architecture: a Comprehensive Review and Directions for Future Research. *Journal of Grid Computing* 15, 295–321 (2017).

[3] Merceedi, K. J. & Sabry, N. A. A Comprehensive Survey for Hadoop Distributed File System. *Asian Journal of Research in Computer Science* 46–57 (2021).

[4] T. M. Cover and P. E. Hart, Nearest neighbor pattern classification, IEEE Transactions on Information Theory, vol. 13, no. 1, pp. 21–27,1967.

[5] Ghazali, R., Adabi, S., Down, D. G. & Movaghar, A. A classification of hadoop job schedulers based on performance optimization approaches. *Cluster Computing* 24, 3381–3403 (2021).

[6] Gandomi A, Reshadi M, Movaghar A, Khademzadeh A. HybSMRP: a hybrid scheduling algorithm in Hadoop MapReduce framework. J Big Data (2019).

[7] Ghazali, R., Adabi, S., Rezaee, A., Down, D. G. & Movaghar, A. CLQLMRS: improving cache locality in MapReduce job scheduling using Q-learning. *Journal of Cloud Computing* 11, (2022).

[8] Luo, Y., Shi, J. & Zhou, S. JeCache: Just-Enough Data Caching with Just-in-Time Prefetching for Big Data Applications. Proceedings - International Conference on Distributed Computing Systems 2405–2410 (2017).

[9] Vinutha, D. C. & Raju, G. T. Data Prefetching for Heterogeneous Hadoop Cluster. 2019 5th International Conference on Advanced Computing and Communication Systems, ICACCS 2019 554–558 (2019).

[10] Lee, J., Kim, K. T. & Youn-Chen, T. MapReduce Performance Scaling Using Data Prefetching, 9, 26–31 (2022).

[11] Kalia, K. et al. Improving MapReduce heterogeneous performance using KNN fair share scheduling. Robotics and Autonomous Systems 157, 104228 (2022).

[12] Dong, B. et al. Correlation-based file prefetching approach for Hadoop. Proceedings - 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010 41–48 (2010).

[13] Singh, G., Chandra, P. & Tahir, R. A Dynamic Caching Mechanism for Hadoop using Memcached (2012)

[14] Chen, Q., Zhang, D., Guo, M., Deng, Q. & Guo, S.: SAMR: a self-adaptive MapReduce scheduling algorithm in a heterogeneous environment. In: Proceedings—10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010. pp. 2736–2743 (2010).

[15] Naik, N.S., Negi, A., Sastry, V.N.: Performance improvement of MapReduce framework in heterogeneous context using reinforcement learning. Procedia Comput. Sci. 50, 169–175 (2015)

[16] Kwak, J., Hwang, E., Yoo, T., Nam, B. & Choi, Y. In-memory Caching Orchestration for Hadoop. (2016).
[17] H. Li, H. Jiang, D. Wang, B. Han, An improved KNN algorithm for text classification, Eighth International Conference on Instrumentation & Measurement, Computer, Communication and Control IMCCC, 2018, pp. 1081–1085. (2018)
[18] TULGAR, T., HAYDAR, A. & ERŞAN, İ. A Distributed K Nearest Neighbor Classifier for Big Data. Balkan Journal of Electrical and Computer Engineering 6, 105–111 (2018).
[19] Maillo, J., Triguero, I. & Herrera, F. A MapReduce-Based k-Nearest Neighbor Approach for Big Data Classification. Proceedings - 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2015 2, 167–172 (2015).
[20] Huang S, Huang J, Dai J, Xie T, Huang B ,The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis (2014).
[21] Hibench, http://GitHub. com/ Intel- bigdata/ HiBench, last accessed 2023/06/25