

FORESEER: Workload-aware Data Storage for MapReduce

Jia Zou†, Juwei Shi†, Tongping Liu‡, Zhao Cao†, and Chen Wang‡

†IEEE Member
‡IBM Research - China
§University of Texas at San Antonio
Email: †jia.zou.us@ieee.org, ‡{jwshi, caozhao, wangcwc}@cn.ibm.com, §tongping.liu@utsa.edu

Abstract—Inter-job Write once read many (WORM) scenario is ubiquitous in MapReduce applications that are widely deployed on enterprise production systems. However, traditional MapReduce auto-tuning techniques cannot address the inter-job WORM scenario. To address the shortcomings in existing works, this paper presents a novel online cross-layer solution, FORESEER. It can automatically predict workloads’ data access information and tune data placement parameters to optimize the over-all performance for an inter-job WORM scenario. In our experiments, we observe that FORESEER can achieve significant performance speedup (up to 37%) compared with previous work.

I. Introduction

Automatic parameter tuning for MapReduce jobs to offer timely and cost-effective big data processing with least effort is a key for success in many businesses, such as banking, telecommunication and etc. [4], [6]. In this paper, we mainly identify and address the problem of automatic parameter tuning for an inter-job write once read many (WORM) scenario. In such scenario, once a MapReduce job (producer) outputs data to a distributed file system (abbreviated as DFS), the data will be processed by one or more MapReduce jobs (consumers) repeatedly. The inter-job WORM scenario is ubiquitous in many popular applications deployed in enterprise and cloud, such as data warehouse and analytic workflows.

MapReduce auto-tuning tools like MRTuner [6] and What-if Engine [3] can not optimize the overall performance for an inter-job WORM scenario, because they focus on performance of a single job. When applied to optimize the producer’s performance, they are not aware of consumers and do not consider the inter-job trade-offs between the producer and consumers.

To address the shortcomings of existing works, we focus on the data placement parameters for the producer’s output, such as block size, number of partitions (i.e. controlled by the number of writing tasks), and replication factor, which are determining the performance trade-offs between the producer and consumers in an inter-job WORM scenario.

We find and prove that a consumer’s data transfer overhead can be minimized, if and only if block size is aligned with the consumer’s map split size, and data is proportionally distributed to each node’s maximum parallelism. Then, we develop a greedy algorithm to search for the optimal data placement parameters, which can efficiently decide block size and number of partitions based on above deduction and significantly reduce the search space.

Another challenge in current practice is that the DFS writing process is unaware of workloads’ data access pattern, which makes it difficult to collect the consumer information required by the greedy algorithm. To solve the problem, we propose a novel cross-layer solution, FORESEER, which can automatically collect runtime behavior of previous jobs. Thus, when a new job is launched, FORESEER can automatically predict consumer information if the same type of jobs have been executed before. Then the greedy algorithm can be applied to compute optimal data placement parameters.

II. Optimization of Data Placement Parameters

To achieve optimal over-all performance in an inter-job WORM scenario, we target to minimize total overhead including the overhead of writing data to storage in the producer and the overhead of transferring data in all consumers’ map phases. We can prove following deduction, with the proof omitted due to space limit: The overhead incurred by a given type of consumer job, can be minimized if and only if data block size is equal to the map split size of that type of consumer job, and data stored in each node is proportional to the number of slots in each node.

Now we develop a greedy algorithm that starts from optimization for the most frequent consumer job, where we can efficiently decide block size and number of partitions based on above deduction. Thus the problem is reduced to searching for the best replication factor.

We denote the size of the producer’s output data by $D$. In addition, we denote other workload-related information by $W$, which includes the number of consumer types $C = \{\text{consumer}_i\}$, occurrence of each consumer type $f_i$, split size of each consumer type $S_i$, index of the most frequent consumer type in the array of consumer types $d$. We also denote platform information by $P$, which includes number of nodes $n$, number of slots $n_s$, memory buffer size for each...
writting task(mb), and other information necessary to com-
pute producer-incurred overhead(O_{producer}), and consumer-
incurred overhead(O_{consumer}). The modeling of O_{producer}
and O_{consumer} is omitted due to space limit. Given above
information collected or predicted, the greedy algorithm can
be formalized as shown in Algorithm 1. The complexity of the
algorithm is linear with the number of nodes in the cluster.

Algorithm 1 Data Placement Optimization for WORM.
Input: D, W, P;
Output: Block Size: B; Partition Number: np; Replication factor: k
1: \( B \leftarrow S_d \)
2: \( nw \leftarrow \frac{D}{nw} \)
3: \( np \leftarrow nw \cdot ns \)
4: \( O_{\min} \leftarrow \infty \)
5: \( k_{\text{optimal}} \leftarrow k_{\text{default}} \)
6: for each \( k_{\text{default}} \leq k \leq n \) do
7: \( O \leftarrow O_{\text{producer}} + \sum_{i=1}^{n} f_i \cdot O_{\text{consumer}} \)
8: if \( O < O_{\min} \) then
9: \( O_{\min} \leftarrow O \)
10: \( k_{\text{optimal}} \leftarrow k \)
11: end if
12: end for
13: \( k \leftarrow k_{\text{optimal}} \)
14: return

III. FORESEER Architecture

Now we propose the design of a cross-layer solution:
FORESEER, which is processing between MapReduce frame-
work and DFS to enable workload-aware data storage. As
shown in Fig. 1, FORESEER is responsible to collect historical
information, predict workload information, and optimize data
placement parameters using the greedy algorithm.

IV. Evaluation

We implement FORESEER in Java for Hadoop 1.1.1. We
setup experimental environment in a cluster consists of ten
HS21 blade servers connected by 10Gb Ethernet. We install
Hadoop 1.1.1 and FORESEER on the cluster. We evaluate
FORESEER using three representative benchmarks from the
HiBench [5]: Terasort, NutchIndex and KMeans.

As shown in Fig. 2, for Terasort, KMeans, and NutchIndex,
FORESEER can reduce the total elapsed time by up to 37%, 30%
and 15% respectively.