2015 IEEE 35th International Conference on Distributed Computing Systems

Aurora: Adaptive Block Replication in Distributed File Systems

Qi Zhang*, Sai Qian Zhang*, Alberto Leon-Garcia*, Raouf Boutaba†
*Department of Electrical and Computer Engineering, University of Toronto
{ql.zhang, sai.zhang, alberto.leongarcia}@mail.utoronto.ca
†David R. Cheriton School of Computer Science, University of Waterloo
rboutaba@uwaterloo.ca

***Abstract*—Distributed file systems such as Google File System and Hadoop Distributed File System have been used to store large volumes of data in Cloud data centers. These systems divide data sets in blocks of fixed size and replicate them over multiple machines to achieve both reliability and efficiency. Recent studies have shown that data blocks tend to have a wide disparity in data popularity. In this context, the naïve block replication schemes used by these systems often cause an uneven load distribution across machines, which reduces the overall I/O throughput of the system. While many replication algorithms have been proposed, existing solutions have not carefully studied the placement of data blocks that balances the load across machines, while ensuring node and rack-level reliability requirements are satisfied.**

In this paper, we study the dynamic data replication problem with the goal of balancing machine load while ensuring machine and rack-level reliability requirements are met. We propose several local search algorithms that provide constant approximation guarantees, yet simple and practical for implementation. We further present Aurora, a dynamic block placement mechanism that implements these algorithms in the Hadoop Distributed File System with minimal overhead. Through experiments using workload traces from Yahoo! and Facebook, we show Aurora reduces machine load imbalance by up to 26.9% compared to existing solutions, while satisfying node and rack-level reliability requirements.

I. Introduction

Large-scale Cloud applications often require significant storage and I/O capacity. To provide scalable and fault-tolerant storage for large volumes of data, distributed file systems such as Google File System (GFS) [14], Hadoop Distributed file system (HDFS) [1] have been recently developed. These systems divide each individual file into multiple fixed-size blocks, and replicate them across a large number of machines in the cluster. Today, distributed file systems have been widely used by Cloud companies such as Google, Amazon, Facebook and Yahoo! to support a large variety of services and parallel computing frameworks such as MapReduce [13], and Spark [21]. Consequently, optimizing the performance of distributed file systems has become a critical concern of today's Cloud service providers.

One of the key issues in the design of distributed file systems is the placement of data blocks. On one hand, the placement of blocks should be fault-tolerant. In other words, the failure of a single node or a Top-of-Rack (ToR) switch should not render a file inaccessable. On the other hand, the placement of blocks should also achieve high I/O efficiency and low replication overhead. By default, the current HDFS replicates each data block 3 times across 2 racks, and the locations of the machines and racks are randomly chosen[1] [2]. While this replication scheme meets the fault-tolerance requirement, it does not necessarily achieve high I/O efficiency. In particular, it has been reported that files in production data centers often have skewed distributions of popularity. Furthermore, these file popularity distributions are subject to change over time. In this context, a constant replication factor of each block may cause the machines that own popular data block to become the performance bottleneck of the cluster, reducing the I/O throughput of the overall file system.

We use MapReduce [3] as an example to illustrate this problem. In a MapReduce job, a map task takes as input a data block stored in the distributed file system, and applies a user-defined map function to produce its output. In this process, if a map task is scheduled on a machine that owns a local copy of the input block, the task is called a *local* task and its execution involves only local disk access. Otherwise, the map task is called a *remote* task because it must fetch the data block from a remote machine. As network I/O is typically slower than local disk access, it has been shown that on average local tasks run $2\times$ faster than remote tasks [20]. Therefore, many recent scheduling algorithms have been proposed to improve data locality [17], [20] so as to reduce the number of remote tasks scheduled in the cluster. However, as file popularity is unevenly distributed in production MapReduce clusters, machines that own popular data blocks can easily become the performance "hotspots" in the cluster, making locality-aware scheduling a difficult challenge.

To address this issue, a technique called *dynamic block replication* has been proposed in the literature [9], [10], whose goal is to replicate data blocks dynamically according to their popularity. By replicating popular blocks across large numbers of machines, the chances of scheduling local tasks can be significantly increased. However, while many block replication heuristics have been proposed in the literature, existing solutions have not carefully studied the *placement* of the replicas in the cluster. The goal of the block placement problem is to (1) achieve a good load balancing across machines

[1]If the block is written by a parallel computing task (e.g. a reduce task of a MapReduce job), the first replica is placed on the local machine, and the remaining replicas are placed on two random machines in a different rack. Otherwise, all 3 replicas are placed on random machines across 2 racks.

1063-6927/15 $31.00 © 2015 IEEE
DOI 10.1109/ICDCS.2015.52

to eliminate performance "hotspots", (2) ensure each block remains available in the presence of node or switch failures. To the best of our knowledge, finding effective block placement algorithms that simultaneously satisfy both requirements is still an unresolved challenge.

In this paper, we study the block placement problem that jointly controls the number of block replicas and their placement in the cluster, with the goal of balancing the load of individual machines while satisfying fault-tolerance requirements. Specifically, we make the following two contributions:

- We analyze the optimal block placement problem from a theoretical perspective. We first show that optimal block placement problem is $\mathcal{NP}$-hard, and then present several constant-factor approximation algorithms for different cases of this problem. Commonly used for solving $\mathcal{NP}$-hard problems, a ρ-approximation algorithm is a polynomial time algorithm that produces a solution that is no worse than ρ times the optimal solution. Our algorithms are based on an optimization framework called *local search*. Starting from an arbitrary initial block placement configuration, our algorithms gradually improve the solution quality until it converges to a near-optimal solution. Our algorithms are adaptive to dynamic conditions, and provides a simple mechanism for balancing the trade-off between solution optimality and reconfiguration cost.
- We show our local search algorithms can be implemented efficiently in distributed file systems such as HDFS. To this end, we design Aurora, a framework for AUtomatic Replication for distributed file StORAage. Aurora uses file popularity information to decide the optimal number of replicas for each block, and gradually update the placement of replicas to achieve near-optimal load balancing while respecting fault-tolerance constraints.

The rest of the paper is organized as follows. We first provide an overview of HDFS in Section II. We then describe several variants of block placement problem, and present a local search approximation algorithm for each variant in Section III. Section IV discusses how our algorithm can achieve trade-offs between solution optimality and reconfiguration cost. We then introduce the design of Aurora in Section V and describe how it implements the proposed algorithms with minimal overhead. Our evaluation using both trace-based simulation and real implementation are presented in Section VI. Finally, we summarize related work in Section VII, and conclude the paper in Section VIII.

II. Background and Related Work

Distributed file systems such as GFS and HDFS are designed to store large volumes of data across a large number of commodity machines. As an open source implementation of GFS, HDFS is the *de facto* storage system for the Apache Hadoop distributed computing framework, supporting a variety of services such as MapReduce, HBase [4] and Spark [21]. In HDFS, each file is partitioned into one or more blocks within a maximum block size, which is set to 64MB by default. In general, except the last block, every block in a file has the size equal to the maximum block size. In practice, the number of

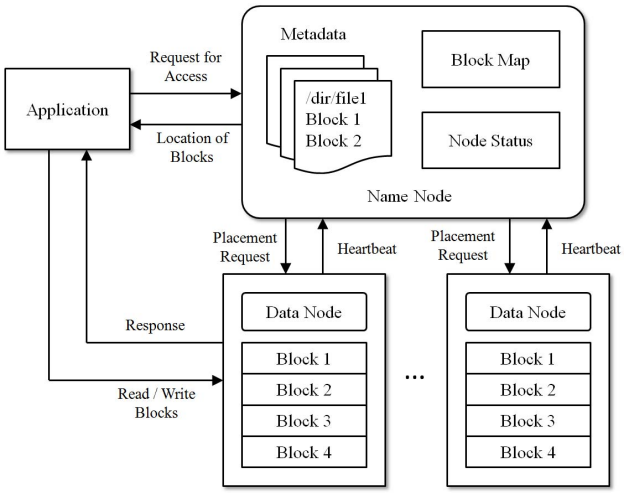


Figure 1. HDFS Architecture

blocks with size less than the maximum block size is usually small. As HDFS is optimized for reading and writing blocks with maximum block size, it is a common practice to minimize the number of small blocks [5].

A HDFS cluster consists of a single *namenode* and multiple *datanodes* running on multiple machines as depicted in Figure 1. The namenode maintains the metadata of the file system, which stores the directory structure, file descriptions and a block map which identifies the location of each block replica in the cluster. Each datanode is responsible for storing the actual data blocks on each machine, and handling incoming read and write requests. Each datanode also periodically sends a heartbeat message to the namenode to report machine and block status. Each application creates a HDFS client to access the file system. A read request can be handled by asking the namenode to provide a machine that owns a replica, and then reading the block from that machine. To handle a write request, a HDFS client first asks the namenode to update the metadata. The namenode responds with a write permission indicating the machine on which the file should be written. The client then writes the file onto the machine, and the datanode of the machine will then request the namenode to provide additional machines for replication. The write process is complete after all replicas of the blocks have been written.

To cope with node and rack switch failures, HDFS replicates each block across multiple nodes in different racks. By default, HDFS replicates each block 3 times across 2 racks [2]. The replication process proceeds by first writing the block to a preferred machine (e.g. local machine) selected by the namenode, and then replicating the block in 2 machines in a remote rack. By default, all the files stored in HDFS have the same replication factor. This default block replication scheme can perform well if block popularity is evenly distributed. However, in practice, it has been observed data blocks tend to have a wide disparity in data popularity. For instance, Abed et. al. [9] shows that file popularity in one of Yahoo!'s MapReduce cluster follows a long-tail distribution. As a result, machines that own popularly data blocks can become the

performance bottleneck of the cluster. Ananthanarayanan et. al. [10] reports that one-sixth of the machines account for half the locality contention in one of the clusters at Microsoft. Therefore, it is desirable to replicate popular blocks and carefully place them in the cluster so as to to balance the load of each individual machine. Even though HDFS provides the API for specifying the replication factor of each file, Currently this must be done manually by the operator. Lastly, while HDFS does provide a balancer tool, its purpose is to balance disk usage rather than machine load.

III. Approximation Algorithms for Block Placement Problem

In this section, we provide a theoretical study of the optimal block placement problem, whose goal is to place file blocks across machines in cluster in order to balance the load of ech individual machine, while meeting block reliability require–ments. Specifically, we study block placement problem for 3 different cases, and provide a local-search approximation algorithm for each case. While some of the cases are already applicable for load balancing the current HDFS (e.g. Section III.B), the main purpose of studying these cases is to grad–ually introduce our technical results for the dynamic block placement and replication algorithms.

In our model, at a given time there are B file blocks stored in the file system. We assume each block $i \in B$ has a popularity score P_i that measures the number of jobs that needs access to the content of block i over a fixed time period T. We further assume block i is replicated k_i times by the file system. The value of k_i can be determined by the reliability requirement of the block [12]. We assume each replica of the block i is identical and has popularity $p_i = \frac{P_i}{k_i}$. In other words, the popularity of block i is shared between its replicas. This means that if we spread the replicas of block i across k_i machines, then the demand for block i can be divided among these k_i machines.

We assume there are M identical physical machines in the cluster that are grouped in R racks. Let $M_r \subseteq M$ denote the machines in rack R. Each machine has a fixed storage capacity. In our model, we define the capacity C_m of a machine m as the maximum number of blocks that can be stored in machine m. As mentioned previously, since most of the blocks in the file system have a size equal to a maximum block size s_{max}, having a capacity C_m for the number of blocks can upperbound the total storage capacity used by the file system on each machine.

A. Block Placement with Known Replication Factor and Node-Level Fault-Tolerance

We first consider the simplest scenario where each block i has a fixed node-level replication factor k_i, and rack-level fault-tolerance is not considered. Let $x_{im} \in \{0,1\}$ as a boolean variable representing whether a replica of block i is placed on machine m. Let $L_m = \sum_{i \in B} p_i x_{im}$ denote the load of machine m based on file popularity. The goal of the replica placement problem is to minimize the maximum load among all the machines in the cluster. It can be represent as the following integer linear program (ILP):

$$\begin{aligned} \underset{x_{im}}{\text{minimize}} \quad & \lambda \\ \text{subject to} \quad & \lambda \geq \sum_{i \in B} p_i x_{im} \quad \forall m \in M \\ & \sum_{i \in B} x_{im} \leq C_m \quad \forall m \in M \\ & \sum_{m \in M} x_{im} = k_i \quad \forall i \in B \\ & x_{im} \in \{0,1\} \quad \forall i \in B, m \in M \end{aligned} \tag{BP-Node}$$

We first prove the following complexity result:

Theorem 1. *BP-Node is $\mathcal{NP}$-hard.*

Proof: We show that BP-Node can be reduced from the parallel machine scheduling problem, which is $\mathcal{NP}$-hard. In the parallel machine scheduling problem, we are given $\mathcal{M}$ identical machines and $\mathcal{N}$ tasks, each with running time t_i for $1 \leq i \leq \mathcal{N}$. We wish to schedule each task on a machine. let σ_m denote the tasks assigned to machine m. The objective of the problem is find a schedule that minimizes the maximum makespan $\sum_{i \in \sigma_{m'}} t_i$, where $m' = \arg\max_{1 \leq j \leq \mathcal{M}} \sum_{i \in \sigma_j} t_i$.

Given a parallel machine scheduling problem, we can construct an instance of BP-Node as follows: We are given $\mathcal{M}$ identical machines, and n file blocks. Each block has popularity equal to t_i. We also set the replication factor k to 1 and $C_j = \sum_{1 \leq i \leq n} t_i$ for each machine j. Clearly, the optimal solution of this problem is exactly the solution of the parallel machine scheduling problem. Since this problem is $\mathcal{NP}$-hard, our problem is $\mathcal{NP}$-hard as well. ■

For BP-Node, we develop an iterative local-search approx–imation algorithm as represented by Algorithm 1. Starting with an arbitrary initial placement configuration where all k_i replicas of each block i are placed, the algorithm improves the quality of the solution in an iterative fashion. Let $L_l = \sum_{i \in B(m)} p_i$ denote the load of a machine $l \in M$. In each iteration, the algorithm identifies the machine with the highest load $m = \arg\max_{l \in M} L_l$, the machine with the lowest load $n = \arg\min_{l \in M} L_l$, and a block i stored on m to execute one of the following operations:

Algorithm 1 Local Search Algorithm for BP-Node

```
loop
  m ← arg max_{l∈M} L_l, n ← arg min_{l∈M} L_l
  if ∃ a Move(m,i,n) or a Swap(m,i,n,j) operation that
  improves solution quality then
    Perform the move
  else return
  end if
end loop
```

- $Move(m,i,n)$: Move block i from m to n
- $Swap(m,i,n,j)$: Find another block j stored on n, and swap i and j.

Let SOL and OPT denote the value of λ produced by Algorithm 1 and in the optimal solution, respectively. Let

$B(m)$ denote the blocks on a machine $m \in M$, and let $p_{max} = \max_{i \in B} p_i$ denote the popularity of the most popular block in the entire HDFS file system. We can prove the following technical result:

Theorem 2. $SOL \leq OPT + p_{max}$.

Proof: Consider the machine pair (m, n) where m has the highest load L_m and n has the lowest load L_n in the cluster. Without loss of generality, we assume $L_m > L_n$. We now consider the operations that either move a block from m to n, or swap two blocks on m and n. Since at the end of Algorithm 1 no operation can improve the solution quality, We can divide the blocks in m into 2 groups:

1) $\mathcal{R}$, which are the blocks owned by both m and n. In this case, moving a block in $\mathcal{R}$ to n reduces its replication factor by 1.
2) $\mathcal{M}$, which are the blocks owned by m but not by n. However, replicating them on n increase the total solution cost.

Since blocks in $\mathcal{R}$ are owned by both m and n and each machine is allowed to have only a single copy of each block, the blocks in $\mathcal{R}$ are already load balanced, i.e. they contribute the same load on each machine. Therefore, we shall focus on the operations involving blocks in $\mathcal{M}$. For the block $i \in \mathcal{M}$ with the highest popularity, (i.e. $i \in \arg\max_{l \in M} p_l$), if $Move(m, i, n,)$ is feasible but not performed, we must have

$$L_m \quad \leq \quad L_n + p_i. \tag{1}$$

Otherwise, if $Move(m, i, n)$ is not feasible, then n must have reached its storage capacity (i.e., n is full). Since m and n both have same capacity and $L_m > L_n$, there must exist blocks in $B(n) \backslash B(m)$ that is owned only by n. To see this, suppose this is false, i.e. m owns all the blocks n has. Since n is full, m must be full as well and therefore $L_m = L_n$ must hold. Since this is not the case, we must have $B(n) \backslash B(m) \neq \{\emptyset\}$.

Now we further argue that there is at least one block $j \in B(n) \backslash B(m)$ such that $p_j \leq p_i$. The reason that, suppose all the blocks in $B(n) \backslash B(m)$ have popularity higher than p_i. Since n is full (i.e. contains at least the same number of blocks as m) and i is the block in $\mathcal{M}$ that has the highest popularity in m, we must have $L_n \geq L_m$, which is a contradiction. Therefore, there must be a block $j \in B(n) \backslash B(m)$ such that $p_j \leq p_i$. In this case, $Swap(m, i, n, j)$ is feasible and reduces L_m. Since it is not performed by Algorithm 1 after it ends, it must be case that n becomes the machine with the high load in the cluster, and it increases the solution cost. Therefore,

$$L_m \quad \leq \quad L_n + p_i - p_j \tag{2}$$

In either of these two cases (Equation (1) and (2)) we have $L_m \leq L_n + p_i$. Since L_n is the machine with the lowest load, we must have $L_n \leq OPT$. Therefore,

$$SOL = L_m \quad \leq \quad L_n + p_i \leq OPT + p_{max}$$

■

Since the block with popularity p_{max} is also scheduled in a machine in OPT, we must have $p_{max} \leq OPT$, therefore we can establish the following result:

Algorithm 2 Local Search Algorithm for BP-Rack

loop
  for $r \in R$ **do**
    $m_r \leftarrow \arg\max_{l \in M_r} L_l$, $n_r \leftarrow \arg\min_{l \in M_r} L_l$
  end for
  if $\exists$ a $Move(m_r, i, n_r)$ **or** $Swap(m_r, i, n_r, j)$ for a any rack $r \in R$, **or** a $RackMove(r, m, i, t, n)$ **or** $RackSwap(r, m, i, t, n, j)$ for any two racks $\{r, t\} \in R$ that improves solution quality **then**
    Perform the move
  else return
  end if
end loop

Corollary 3. *Algorithm 1 is a 2-approximation algorithm for BP-Node.*

B. Load Balancing with Known Replication Factor

The block placement problem studied in the previous session does not consider rack-level fault-tolerance requirement. In our second case of the block placement problem, we want to ensure that each block i is replicated at least k_i times across ρ_i racks. Define y_{ir} as a boolean variable that indicates whether block i has a replica in rack $r \in R$. This variant of the replica placement problem can be represented as the following ILP:

$$\begin{aligned}
\min_{x_{im}, y_{ir}} \quad & \lambda \\
\text{s. t.} \quad & \lambda \geq \sum_{i \in B} p_i x_{im} && \forall m \in M \\
& \sum_{i \in B} x_{im} \leq C_m && \forall m \in M \\
& \sum_{m \in M} x_{im} = k_i && \forall i \in B \\
& y_{ir} \geq x_{im} && \forall i \in B, m \in M_r \\
& \sum_{r \in R} y_{ir} \geq \rho_i && \forall i \in B \\
& x_{im}, y_{ir} \in \{0, 1\} && \forall i \in B, m \in M
\end{aligned} \tag{BP-Rack}$$

It is easy to see that the problem generalizes the problem studied in the previous section. To solve this problem, we introduce two additional operations as follows:

- $RackMove(r, m, i, t, n)$: Move a block i from a machine m in a rack r to a machine n in a rack t
- $RackSwap(r, m, i, t, n, j)$: Find a block i stored on machine m in a rack r and a block j stored on n, and swap these two blocks.

Let SOL and OPT denote the value of λ produced by Algorithm 1 and in the optimal solution, respectively. We can prove the following results:

Theorem 4. $SOL \leq OPT + 3p_{max}$.

Proof: Let R_{max} and R_{min} denote the racks with maximum load and minimum load. When Algorithm 2 finishes, since no $Move(m, i, n)$ or $Swap(m, i, n, j)$ can improve the

solution cost, each rack must be load-balanced. Specifically, let m_{max} denote the machine in R_{max} having the highest load $L_{m_{max}}$ By Theorem 1, we have $L_{m_{max}} \leq L_m + p_{max}$ for any machine $m \in R_{max}$. Similarly, let n_{min} denote the machine in R_{min} having the lowest load $L_{n_{min}}$. By Theorem 1, we have $L_n \leq L_{n_{min}} + p_{max}$ for any any machine $n \in R_{min}$.

We now consider the blocks stored in racks R_{max} and R_{min}. If both R_{max} and R_{min} have the same load, then all the racks must have the same load, thus by Theorem 1, balancing machine in R_{max} already achieves $SOL = L_{m_{max}} \leq OPT + p_{max}$. Therefore, we focus on the case where rack R_{max} has strictly higher load than R_{min}, i.e., $\sum_{i \in R_{max}} L_i > \sum_{i \in R_{min}} L_i$. The blocks stored in rack R_{max} can be divided into 4 groups:

1) $\mathcal{R}$, which are the blocks owned by machines in R_{max} and R_{min}, however, each block $i \in \mathcal{R}$ is replicated in ρ_i racks, and each rack R_{max} and R_{min} only owns a single copy of the block. In this case, moving the blocks in $\mathcal{R}$ to machines in R_{min} reduces their rack-level replication factors by 1.
2) $\mathcal{L}$, which are the blocks owned by machines in R_{max} and R_{min}, each block $i \in \mathcal{R}$ is replicated in ρ_i racks. Machines in R_{max} only owns a single copy of the block, but machines in R_{min} owns multiple copies of the block. Similar to $\mathcal{R}$, moving the blocks in $\mathcal{L}$ to machines in R_{min} reduces the rack-level replication factors by 1.
3) $\mathcal{P}$, which are the remaining blocks owned by both R_{max} and R_{min}. However, these blocks are freely moveable from machine in R_{max} to machines in R_{min} without reducing their rack-level replication factors.
4) $\mathcal{M}$, which are the blocks owned only by machines in R_{max} but not by machines R_{min}.

Since blocks in $\mathcal{R}$ are owned by both R_{max} and R_{min}, and only a single copy is available in each rack, the blocks in $\mathcal{R}$ are already load balanced across the two racks. Since blocks in $\mathcal{L}$ are not moveable, We shall focus on moving blocks in $\mathcal{P} \cup \mathcal{M}$. We first argue that there is at least one block in $\mathcal{P} \cup \mathcal{M}$. The reason is that if $\mathcal{P} \cup \mathcal{M} = \{\emptyset\}$, since blocks in $\mathcal{R}$ is already load balanced, and blocks in $\mathcal{L}$ has more copies in rack R_{min}, then R_{min} must have higher load than R_{max}, which is a contradiction. Now, let $i \in \arg\max_{l \in \mathcal{P} \cup \mathcal{M}} p_l$ denote the block with the highest popularity in $\mathcal{P} \cup \mathcal{M}$. Let $m \in R_{max}$ denote a machine that owns block i. Suppose there exists a machine $n \in R_{min}$ that is not full. Since at equilibrium, $Move(m, i, n)$ is feasible but not performed, we must have

$$L_m \quad \leq \quad L_n + p_i. \tag{3}$$

Otherwise, it must be the case that all the machines in R_{min} are full. Let $\bigcup_{j \in R} B(j)$ denote the blocks owned by machines in rack R. In this case, since $\sum_{i \in R_{max}} L_i > \sum_{i \in R_{min}} L_i$, there must exist blocks in $\bigcup_{j \in R_{max}} B(j) \backslash \bigcup_{j \in R_{min}} B(j))$ that is owned only by machines in rack R_{min}. To see this, suppose this is false, i.e. R_{max} owns all the blocks R_{min} owns. Since R_{min} is full, R_{max} must be full as well and therefore $\sum_{j \in R_{max}} L_j = \sum_{j \in R_{min}} L_j$ must hold. Since this is not the case, we must have $\bigcup_{j \in R_{max}} B(j) \backslash \bigcup_{j \in R_{min}} B(j)) \neq \{\emptyset\}$. Notice that $\mathcal{L}$ also belongs to this group.

Now we further argue that there is at least one block j owned by a machine in Rack $R_m in$ such that $p_j \leq p_i$. The reason that, suppose all the block replicas in $\bigcup_{j \in R_{min}} B(j) \backslash \bigcup_{j \in R_{max}} B(j))$ have popularity strictly higher than p_i. Since all the machines in R_{min} is full (i.e. contains least the same number of blocks as m) and i is the block in $\mathcal{P} \cup \mathcal{M}$ that has the highest popularity in m, and rack R_{min} has more copies of block in $\mathcal{L}$ than R_{max}, we must have $\sum_{i \in R_{max}} L_i \leq \sum_{i \in R_{min}} L_i$, which is a contradiction. Therefore, there must exist a block $j \in \bigcup_{j \in R_{min}} B(j) \backslash \bigcup_{j \in R_{max}} B(j))$ such that $p_j \leq p_i$. In this case, $RackSwap(R_{max}, m, i, R_{min}, n, j)$ is feasible and reduces L_m. Since it is not performed by Algorithm 2 after it ends, it must be case that n becomes the node with the high load in the cluster, and it increases the solution cost. Therefore,

$$L_m \quad \leq \quad L_n + p_i - p_j \tag{4}$$

In either case (Equation (3) and (4)) we have $L_m \leq L_n + p_i$. Since $L_{n_{min}}$ is the machine with the lowest load in rack R_{min}, we must have $L_{n_{min}} \leq OPT$ and therefore

$$\begin{aligned} L_{m_{max}} &\leq L_m + p_{max} \leq L_n + p_i + p_{max} \\ &\leq L_{n_{min}} + p_{max} + p_i + p_{max} \\ &\leq L_{n_{min}} + 3p_{max}. \end{aligned}$$

■

Similar to the previous case, since the block with popularity p_{max} is also scheduled in a machine in OPT, we must have $p_{max} \leq OPT$. Therefore we have the following result:

Corollary 5. *Algorithm 2 is a 4-approximation algorithm for the BP-Rack.*

C. Replication Factor Aware-Scheduling

In the third case of the block placement problem, we allow the file system to choose the number of replicas for each block, as long as the minimum machine-level and rack-level fault-tolerance requirements for each block are satisfied. Similar to [10], we assume there is a total replication budget β that represents that maximum storage capacity for storing and replication blocks in B, the optimization problem becomes

$$\begin{aligned} \min_{x_{im}, y_{ir}} \quad & \lambda \\ \text{s. t.} \quad & \lambda \geq \sum_{i \in B} \frac{P_i}{k_i} x_{im} && \forall m \in M \\ & \sum_{i \in B} x_{im} \leq C_m && \forall m \in M \\ & \sum_{m \in M} x_{im} \geq k_i && \forall i \in B \\ & y_{ir} \geq x_{im} && \forall i \in B, m \in M_r \\ & \sum_{r \in R} y_{ir} = \rho && \forall i \in B \\ & k_i \geq k_{low} && \forall i \in B \\ & \sum_{i \in B} k_i \leq \beta \\ & x_{im}, y_{ir} \in \{0, 1\} && \forall i \in B, m \in M \end{aligned} \tag{BP-Replicate}$$

To solve BP-Replicate, we first prove the following result:

Theorem 6. *Suppose we replicate each block i k_i times, where $k_i, i \in B$ is the solution of the following optimization problem:*

$$\begin{aligned} \underset{k_i}{\textit{minimize}} \quad & \omega \\ \textit{subject to} \quad & \omega \geq \frac{P_i}{k_i} \quad \forall i \in B \\ & |M| \geq k_i \geq \bar{k}_i \quad \forall i \in B \\ & \sum_{i \in B} k_i \leq \beta \end{aligned} \tag{Rep–Factor}$$

The resulting placement after running Algorithm 2 is a 4-approximation of BP-Replicate.

Proof: Let $\{k_1, k_2, ..., k_{|B|}\}$ denote the optimal solution of the above problem, and let ω denote the cost of the optimal solution. Furthermore, let k_i^* denote the optimal replication factor of i. It is easy to see that $\{k_1^*, k_2^*, ..., k_{|B|}^*\}$ is also a feasible solution of the optimization problem above. Let ω^* denote the cost of this solution. Clearly, since ω is the optimal solution, we must have $\omega \leq \omega^*$. It is clear $\omega^* \leq OPT$ since the optimal solution must contain the block with popularity ω^*. Now consider the solution of Algorithm 2 after setting replication factor to $\{k_1, k_2, ..., k_{|B|}\}$. Theorem 3 states that

$$L_{m_{max}} \quad \leq \quad L_{n_{min}} + 3\omega. \tag{5}$$

Observe that $L_{n_{min}} \leq OPT$. This is because the total load in the cluster is precisely $\sum_{i \in B} P_i$ in both optimal solution and the solution produced by Algorithm 2. By definition, we must have $OPT \geq \frac{1}{|M|} \sum_{i \in B} P_i$ and $L_{n_{min}} \leq \frac{1}{|M|} \sum_{i \in B} P_i$ since n_{min} is the node with the lowest load in the cluster. Combining these equations, we have $L_{n_{min}} \leq OPT$. Substituting $\omega \leq OPT$, we have $SOL = L_{m_{max}} \leq OPT + 3OPT = 4OPT$. ■

Now we are left with the problem of solving Rep–Factor. Algorithm 3 is our solution algorithm. At every iteration, it selects the block with the highest per–replica popularity and tries to increase its replication factor by 1. If doing so exceeds the replication budget β, we try to find a block whose replication factor can be reduced without violating machine–level fault–tolerance requirement, and reduce the block replication factor by 1. This algorithms terminates when the maximum per–block popularity can no longer be reduced without violating the replication budget. The following results establish its optimality:

Lemma 7. *For any solution to the optimization problem where constraint $\sum_{i \in B} k_i < \beta$ is not tight, there exists an equivalent solution with no higher cost for which $\sum_{i \in B} k_i = \beta$.*

Proof: If $\sum_{i \in B} k_i < \beta$ in the optimal solution, we can replicate any blocks until $\sum_{i \in B} k_i = \beta$. Clearly, the cost of this solution is no greater than the cost of the optimal solution. ■

Theorem 8. *Algorithm 3 solves Rep-Factor optimally.*

Proof: Let $i = \max_{j \in B} \{\frac{P_j}{k_j}\}$ produced by Algorithm 3. Let k_i denote the replication factor if i in our solution. In this case, our solution cost produced by Algorithm 3 is precisely $\frac{P_i}{k_i}$. Let k_i^*denote the replication factor of i in the optimal solution where $\sum_{i \in B} k_i = \beta$. According to Lemma 4, this solution always exists. Clearly, if $k_i^* \leq k_i$, then $\frac{P_i}{k_i} \leq \frac{P_i}{k_i^*}$ and our solution must be optimal. Thus, we concentrate on the case where $k_i^* > k_i$. Since $\sum_{i \in B} k_i = \beta$ is satisfied, there must exist another block j such that $k_j^* < k_j$. Now, consider the operation where we increase k_i by 1 and decrease j by 1. This is clearly a feasible move. Since Algorithm 3 does not perform this move, it must be the case that the solution cost is increased and therefore $\frac{P_i}{k_i} \leq \frac{P_j}{k_j - 1}$. Using $k_j^* < k_j$, we must have $\frac{P_i}{k_i} \leq \frac{P_j}{k_j - 1} \leq \frac{P_j}{k_j^*} \leq OPT$, which implies that our solution must be the optimal solution. ■

Algorithm 3 Algorithm for Computing Replication Factor

```
while done = false do
  Find a block i = max_{j∈B} {P_j / k_j}
  if Σ_{j∈B} k_j < β then
    k_i ← k_i + 1
  else if ∃ a block l ≠ i s. t. k_l > k_i^low and P_l/(k_l − 1) ≤ P_i/k_i
  then
    k_l ← k_l − 1, k_i ← k_i + 1
  else
    done ← true
  end if
end while
```

Combine Theorem 6 and 8, we can see that first solving Rep–Factor by Algorithm 3 followed by running Algorithm 2 yields a 4–approximation solution for BP–Replicate.

IV. Trading Solution Optimality for Reconfiguration Cost

So far our analysis has been focusing on solution quality. In practice, moving and swapping blocks can incur a block movement overhead in terms of bandwidth consumption and replica availability. To handle this issue, we take advantage of the fact the local search algorithms allow us to trade solution optimality for migration cost. Specifically, given a real value $\epsilon > 0$, we define a local search operation to be *admissible* if it reduces solution cost by at least $\epsilon \cdot SOL$, where SOL is the cost of the current solution. In other words, a local search operation is performed only if it can substantially reduce the cost the solution. Follow the standard approach (e.g. [16]), We can show these algorithms terminate in polynomial time and deliver constant approximation factors:

Theorem 9. *If only admissible move or swap operations are performed in Algorithm 2 and 3, then each of the algorithms terminates in $O(\frac{1}{\log(1-\epsilon)} \log \frac{SOL}{OPT})$ iterations, where SOL and OPT denote the cost of the initial solution and the optimal solution respectively. In this case, Algorithm 2 and 3 achieve approximation factors of $2 + \epsilon$ and $4 + 3\epsilon$ respectively.*

Proof: The approximation factors of these algorithms can be proven by adding ϵ to the right side of equation (1), (2), (3) and (4). For running time of the algorithms, notice that after each operation the solution cost is reduced by a factor of $1 - \epsilon$.

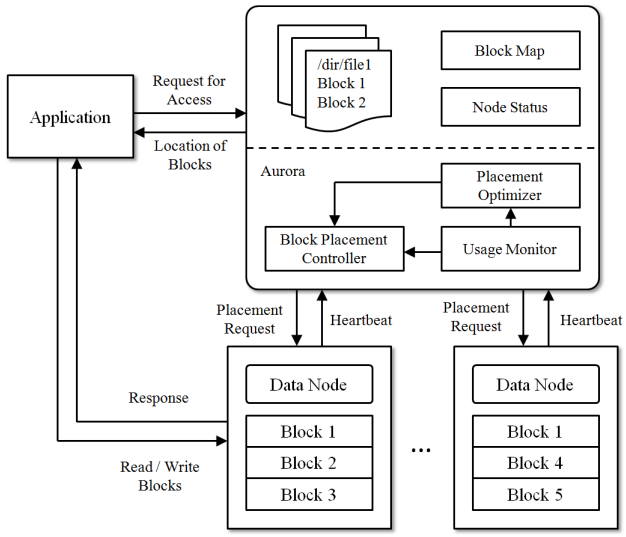


Figure 2. Architecture of AURORA (To be updated)

Therefore, the number of local search operations performed is at most $\log_{1-\epsilon} \frac{SOL}{OPT} = \frac{1}{\log(1-\epsilon)} \log(\frac{SOL}{OPT})$. ■

Thus, we can control the value of ϵ to achieve a tradeoff between solution quality and replication overhead. We quantify this tradeoff experimentally in Section VI.

V. AURORA: A DYNAMIC BLOCK PLACEMENT AND REPLICATION FRAMEWORK FOR HDFS

While we have proposed algorithms that provide theoretical performance guarantee for block placement problem, these algorithms are not directly applicable to dynamic block placement. This is because our algorithms are designed for static instances of the problem, whereas in reality the blocks can be created, accessed and deleted over time. Block popularities can also change dynamically. Therefore, it is necessary to design a system that can adapt block placement according to system dynamics, while incurring minimum reconfiguration overhead.

To this end, we designed Aurora, a system for dynamic block placement in distributed file systems. The design goal of Aurora is to leverage existing HDFS mechanisms as much as possible for block placement and replication. The architecture of Aurora is shown in Figure 2. The *usage monitor* is responsible for collecting usage statistics of individual blocks. The *block placement controller* is responsible for handling block placements. Periodically, the *placement optimizer* uses the statistics information collected by the usage monitor to optimize the placement of blocks through block migration and replication.

Similar to existing work [9], [10], the usage monitor in Aurora determines block popularity by recording the number of accesses of a block within a sliding time window W (i.e. the number of recent accesses in W hours). The exact value of W can be controlled by the operator. While many algorithms (e.g. ARIMA [11]) may be used to predict file popularity in future time periods, we found using the historical value is sufficient to produce high quality solutions for the dynamic block placement problem.

Algorithm 4 Algorithm for Initial Block Placement

if block i is written by a task **then**
  replicate i on local machine
else
  replicate i on machine with lowest load in the rack with the lowest total load
end if
for $j = 2$ to ρ_i **do**
  replicate i on machine with lowest load in the jth rack with the lowest total load
end for
for $j = \rho_i + 1$ to k **do**
  replicate i on the jth machine with lowest load among the top ρ_i racks with the lowest total load
end for

Algorithm 5 Algorithm for Optimizing Block Replication

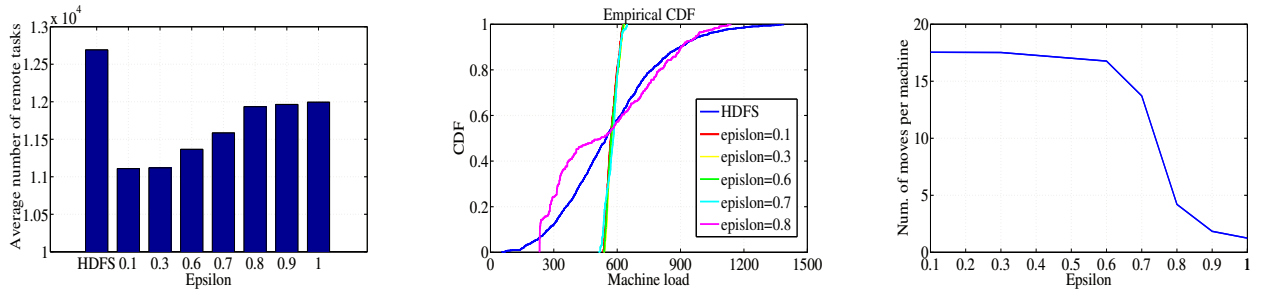
At beginning of a time period t:
$l \leftarrow K$
while $\exists$ a replication move for under replicated block i identified by Algorithm 3 between machine m and n **and** $l > 0$ **do**
  copy block i from m to n, $l \leftarrow l - 1$
end while
while $\exists$ an admissible $Move(m_r, i, n_r)$ **or** $Swap(m_r, i, n_r, j)$ for a rack $r \in R$, **or** a $RackMove(r, m, i, t, n)$ **or** $RackSwap(r, m, i, t, n, j)$ for two racks $\{r, t\} \in R$ **do**
  Perform the move
end while

As represented by Algorithm 4, the block placement controller handles the initial block placement using a greedy approach. Given a block i with node-level replication factor k_i and rack-level replication factor ρ_i, if the block is written by a task, then first replica is written to the local machine that runs the task. Otherwise, it is written to the machine with lowest load in the rack with the lowest total load. The next $\rho_i - 1$ replicas are written to machines in the next $\rho_i - 1$ racks with the lowest total load. The remaining replicas are written among the machines in the $\rho_i - 1$ racks in ascending order of machine load.

The placement optimizer is responsible for optimizing the placement of blocks periodically. As described by Algorithm 5, at a beginning of a period, the optimizer examines the block popularity provided by the usage monitor. Based on this information, the placement optimizer replicates blocks according to Algorithm 3, and attempt to balance the load across the machines. To reduce the replication overhead, we can limit the maximum number of iterations in Algorithm 3 to a constant K, which is a tunable parameter. The replication overhead can be further reduced using techniques such as compression [10], or use remote map tasks to facilitate block replication by writing remote blocks to local machine once they are read [9].

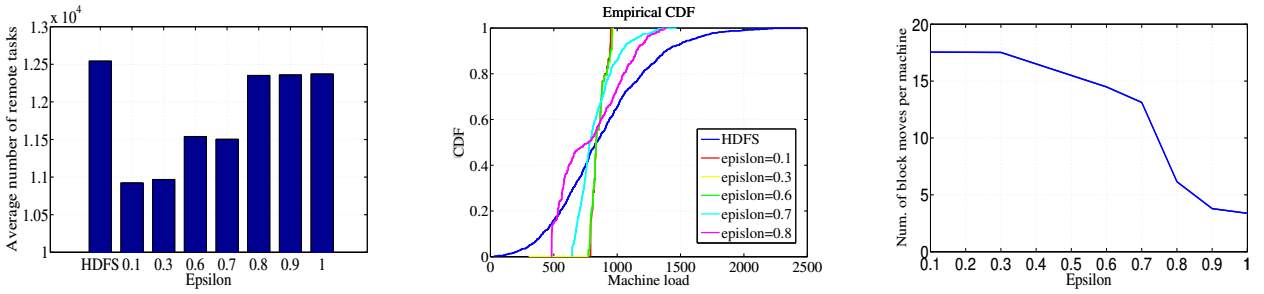
Lastly, deletion of local block replicas is done lazily when disk space is needed. This reduces the disk overhead while allowing Aurora to reclaim the block if the replication factor needs to be increased again.

In summary, Aurora dynamically adjusts block replication factor based block access statistics. When a block needs to be



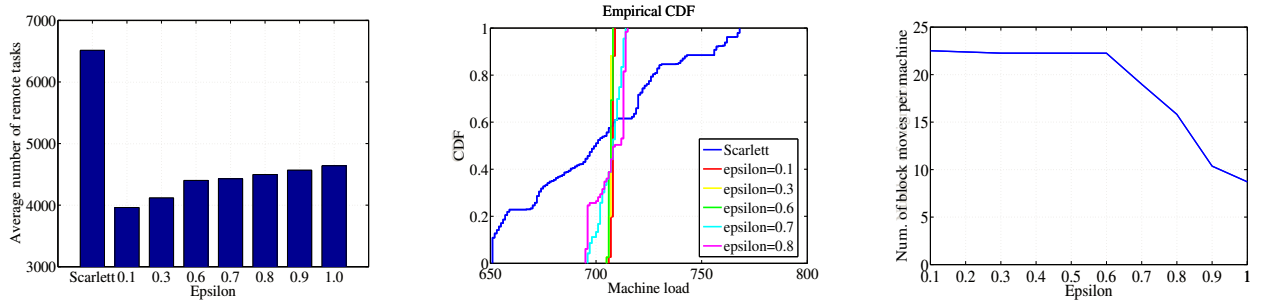
(a) Number of remote tasks (b) CDF of average machine load (c) Number of Block Movements per Hour

Figure 3. Evaluation for Case 1 of the Block Placement Problem



(a) Number of remote tasks (b) CDF of average machine load (c) Number of Block Movements per Hour

Figure 4. Evaluation for Case 2 of the Block Placement Problem



(a) Number of remote tasks (b) CDF of average machine load (c) Number of Block Movements per Hour

Figure 5. Evaluation for Case 3 of the Block Placement Problem

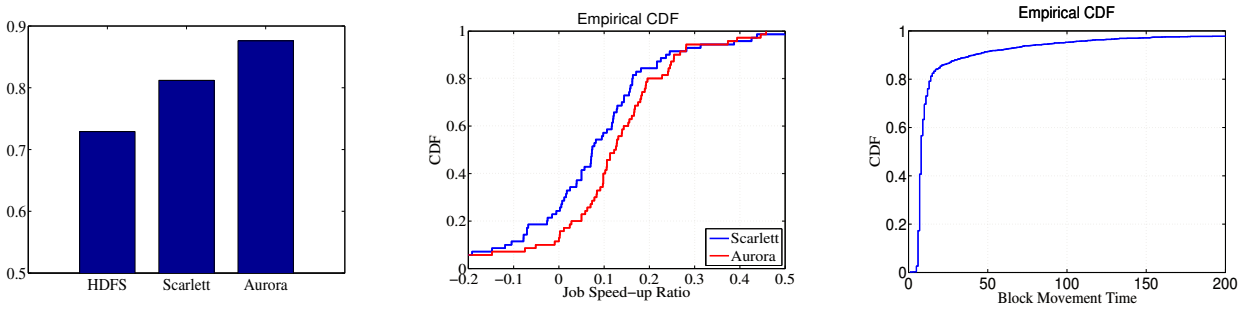
written to HDFS, Aurora minimizes the initial block placement cost. It also optimizes the block placement periodically. Finally, it is evident that if the block usage pattern become stable, over time Aurora will eventually converge to a near optimal solution, as indicated by Theorem 9.

VI. Experiments

We have evaluated the performance of Aurora using both trace-based simulations and real deployment in a 10-node Hadoop testbed. Trace-based simulations allow us to study the effectiveness of Aurora at production scale, whereas testbed deployment allows us to evaluate the performance of Aurora in terms of job completion time and overhead associated with dynamic block movements (i.e. block migration and replication).

A. Trace-based Simulations

We performed traced-based simulation using workload traces provided by Yahoo! [6]. We simulated a cluster of 845 machines distributed over 13 racks. Each rack is identical and contains 65 machines. In our experiment, the mean number of blocks per file is set to 8, and each block is replicated 3 times. In our simulations, each machine has sufficient resources for scheduling 14 tasks simultaneously. For our simulations, we have implemented the random placement algorithm used by the current HDFS, as well as Aurora described in Section V. In our experiments, we set the reconfiguration period to 1 hour, $K = 20000$ and $W = 2$. For comparison purpose, we have also implemented Scarlett [10], which is one of the most well-known block replication schemes found in the literature. The main difference between Scarlett and Aurora is that Scarlett is only designed for block replication, and does not



(a) Percentage of remote tasks

(b) Job speed-up ratio comparison

(c) Movement overhead

Figure 6. Result of the Testbed Experiment

consider initial block placement and dynamic load balancing. Furthermore, it proposes two heuristics, priority and *round robin*, for computing block replication factors. We compare Aurora with *priority*, which achieves better performance than *round robin* in experiments.

We first evaluated the performance of our algorithms for all 3 cases studied in Section III. In the first two cases where no replication is involved, we compare Aurora with the random block placement policy in HDFS using different values of ϵ. In the last case, we compare Aurora with Scarlett using different values of ϵ. The results for the first two cases are shown in Figure 3 and 4 respectively. In the first case where replication factor is set to 3 and rack-level reliability is not considered, we found Aurora can reduce the number of remote tasks by up to 12.5% when $\epsilon = 0.1$, as shown in Figure 3(a). The benefit mainly comes from the even distribution of machine load, as shown in Figure 3(b). Figure 3(c) shows the average number of block movements in the cluster. It is evident that ϵ provides a means to balance the trade-off between load balancing and block movement overhead. Clearly, Aurora achieves a good load balancing when $\epsilon \leq 0.6$, at the cost of high block movement overhead. We can reduce the block movement overhead by increasing ϵ to 0.8. However, in this case the gain in load balancing drops significantly. Such high block movement overhead is also reported in [10], which shows that the block movement overhead can increase network traffic by up to 24%. However, using appropriate compression schemes can reduce the network traffic due to block movement by $27\times$, making the overhead acceptable. Similarly, if compression scheme is used, the amount of network traffic generated by block movement can drop to 0.6 block per machine per hour, which becomes reasonable.

We repeated the same experiment for the second case where rack-level reliability is considered. Our evaluation results are shown in Figure 4. Similar to the first case, we found setting $\epsilon = 0.7$ provides a decent tradeoff between load balancing and block movement overhead. In this case, Aurora improves data locality by 8%, while achieving a block movement overhead of 0.5 block per machine per hour if compression is used.

From the first two experiments, it seems that achieving good load balancing in Aurora incurs a high overhead. However, this is no longer the case when dynamic replication is used. In our last experiment, we compare the performance of Aurora with Scarlett. We set our replication budget β to 70000 additional blocks. The results are shown in Figure 5. Comparing Figure 5(a) and 4(a), we first notice that Scarlett already delivers substantial improvement (12600 remote tasks versus 6500 remote tasks per hour) in terms of reducing the number of remote tasks. At the same time, Aurora still performs better than Scarlett. In all cases, Aurora can further reduce the number of remote tasks by 26.9%, as shown in Figure 5(a). Figure 5(b) shows that Aurora achieves almost perfect load balancing. In terms of block movements, we find that we can find a good trade-off set ϵ to values close to 1, which incurs a block movement overhead of 0.41 block per machine per hour if compression is used.

B. Testbed Evaluation

We have also implemented Aurora in Hadoop 2.5.2 and deployed it in our Hadoop Cluster that consists of 10 nodes, each with 4 virtual CPUs, 8 GB RAM and 320 GB disk space. The implementation of Aurora in HDFS is straightforward. The current HDFS already provides the API to control the number of replicas of each block at run-time. Thus, our main goal was to implement a load-aware block placement policy. We modified the `BlockMap` in HDFS so that it records the popularity of individual blocks. Based on that, the namenode determines the load of each machine, and use this information to replicate blocks. Finally, we implemented the load balancing module based on the existing balancer provided by Hadoop. Different from our load balancer, the existing load balancer only moves blocks to balance the disk usage across machines. Therefore, our load balancer is more powerful than the existing load balancer because we are able to consider both disk usage and block popularity through the use of move and swap operations.

The workload we used for our evaluation is generated using the Statistical Workload Injector for MapReduce (SWIM) [7]. The SWIM Workload repository contains workload traces from Facebook for a 600-node MapReduce cluster. In our experiment, we used SWIM to scale-down the workload so it runs in our testbed. We evaluated the performance of all 3 systems: Default HDFS, Scarlett and Aurora. We used the capacity scheduler for Hadoop Yarn MapReduce [8] for all three systems. In our experiment, we set $\epsilon = 0.8$ as suggested by our simulations. Figure 6(a) shows the percentage of local

tasks achieved with each system. It is evident that Aurora achieves higher task locality compared to both HDFS and Scarlett. Figure 6(b) shows the speedup improvement compared to Scarlett. For each job in the workload, the speedup improvement is determined as the ratio between the reduction in running time and job running time using Scarlett. We find Aurora achieves an average speedup factor of 15% and outperforms Scarlett by up to 8%. We believe this gain will be higher if larger clusters are used, as data locality tends to decrease as the number of machines increases. Figure 6(c) shows the average block movement duration as reported by HDFS. We observe that most of block movements take less than 10 seconds to complete, which is small considering reconfiguration is performed once every hour. Finally, the average number of replication performed is 96 blocks per hour and the number of block movement involved is 10 blocks per hour, which is acceptable even when compression is not used.

VII. Related Work

To cope with skewed distribution of file popularity, many techniques have been proposed in the literature. For instance, CDRM [19] is a system that computes block availability and block probability and use this information to guide the placement of blocks. DARE [9] replicates popular blocks with a probably p after each read access. Unpopular blocks are evicted according to a least-recently used (LRU) policy. However, DARE does not consider the placement of blocks in the system. Scarlett [10] is a system that replicates blocks dynamically based on load distribution. Compare to Scarlett, our algorithms require less input parameters, yet computes the optimal replication factors for each block. While Scarlett provides a simple heuristic for block placement, it is designed primarily for block replication, whereas in our work we design placement algorithms for all common usage scenarios.

Many recent studies have also applied coding theory to achieve high file availability [15]. While these techniques can achieve high storage efficiency and fault-tolerance, they are mainly used for long-term storage, and are often inefficient for parallel processing due to the overhead of decoding files. Finally, file placement problems have also been studied in area of Grid computing (e.g. [18]). However, the problem context they considered is different. For instance, rack-level fault tolerance is not considered in these studies.

VIII. Conclusion

With the rising popularity of big-data analytics in recent years, designing high performance yet fault-tolerant file storage have become a critical challenge. In particular, the naïve block replication schemes implemented in existing systems have not taken data popularity into consideration, which often leads to uneven load distribution across machines. While many heuristics have been proposed for this problem, these heuristics have not studied the placement of blocks in the cluster in order to balance the load across machines, while satisfying machine-level and rack-level reliability requirements.

In this paper, we study the dynamic block replication problem with the goal of balancing the load of individual machines while ensuring machine and rack-level reliability requirements are met. Since the optimal block placement problem is $\mathcal{NP}$-hard, we proposed several constant-factor local search approximation algorithms that are adaptive to dynamic conditions, yet simple for implementation. To this end, We present Aurora, a dynamic block distribution mechanism that implements these algorithms in Hadoop Distributed File System with minimal overhead. Through experiments using real workload traces from Yahoo! and Facebook, we show our approach outperforms existing solutions in terms of load distribution, while ensuring reliability requirements are satisfied. In the future, we are interested in implementing techniques such as replication on read [9] and compression [10] for dynamic block replication.

Acknowledgement

This work was completed as part of the Smart Applications on Virtual Infrastructure (SAVI) project funded under the National Sciences and Engineering Research Council of Canada Strategic Networks grant number NETGP394424-10.

References

[1] Hadoop Distributed File Systems. http://hadoop.apache.org/docs/hdfs/.
[2] HDFS Architecture, https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.
[3] Hadoop MapReduce Distribution. http://hadoop.apache.org.
[4] Apache HBase. http://hbase.apache.org/.
[5] The Small Files Problem, http://blog.cloudera.com/blog/2009/02/the-small-files-problem/.
[6] S3 – Yahoo! Hadoop grid logs, version 1.0. http://webscope.sandbox.yahoo.com /catalog.php?datatype=s.
[7] Statistical Workload Injector for MapReduce (SWIM), https://github.com/SWIMProjectUCB/SWIM/wiki.
[8] Apache Hadoop NextGen MapReduce (YARN), http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.
[9] Cristina L Abad, Yi Lu, and Roy H Campbell. Dare: Adaptive data replication for efficient cluster scheduling. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 159–168, 2011.
[10] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, and Duke Harlan. Scarlett: coping with skewed content popularity in mapreduce clusters. In *EuroSys*. ACM, 2011.
[11] G. E. P. Box, G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis, Forecasting, and Control*. Prentice-Hall, third edition, 1994.
[12] Chris X Cai, Cristina L Abad, and Roy H Campbell. Storage-efficient data replica number computation for multi-level priority data in distributed storage systems. In *Dependable Systems and Networks Workshop*, 2013.
[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.
[14] Sanjay Ghemawat et al. The google file system. In *ACM SOSP*, 2003.
[15] Yadi Ma, Thyaga Nandagopal, Krishna PN Puttaswamy, and Suman Banerjee. An ensemble of replication and erasure codes for cloud file systems. In *IEEE INFOCOM, 2013*.
[16] Tardos Eva Pal, Martin and Tom Wexler. Facility location with nonuniform hard capacities. *Proceedings of FOCS*, 2001.
[17] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: Locality-aware resource allocation for mapreduce in a cloud. 2011.
[18] Kavitha Ranganathan and Ian Foster. Identifying dynamic replication strategies for a high-performance data grid. In *GRID 2001*.
[19] Qingsong Wei, Bharadwaj Veeravalli, Bozhao Gong, Lingfang Zeng, and Dan Feng. Cdrm: A cost-effective dynamic replication management scheme for cloud storage cluster. In *Intl. Conf. on Cluster Comp.*, 2010.
[20] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *ACM Eurosys*, 2010.
[21] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *USENIX HotCloud workship*, 2010.