Deduplication in container image storage system: Model and Cost-Optimization Analysis

1.Problem

Containers have recently gained a significant traction because of their lightweight isolation and fast deployment[1,2]. With the emerge of container orchestration system, platform-as-a-service (PaaS) has been taken by container-as a service (CaaS). Container image has become the way to package up applications and container registries have become the way to distribute and ship applications. Many container registries are built, such as Docker Hub, google domain registry, IBM bluemix, Amazon elastic container registry, Azure Container Registry, GitLab Container Registry[3,4].

Currently, the number of container images is growing rapidly. This massive amount of container images present challenges for the backend storage systems. Open source container registry supports local file system and many kinds of distributed storage systems, such as s3, azure, swift, oss, and gcs as backend storage system. After analyzing ~50TB container images downloaded from Docker Hub, we found there were a lot of file duplicates across different container images. Container images are similar with VM images in the sense that they all serve as OS snapshots. Difference users might choose similar OS and run similar applications, which incurs a considerable redundancy across different container images. Deduplication technique is a good solution to remove the redundancy from container image storage system. However, current deduplication techniques cannot be directly applied on container image storage system because container images are stored as a number of gzip compressed tar files, known as layers. Gzip compressed files have very less deduplication ratio. Moreover, container service requires high startup speed, which puts a tight constraint on image pull time. However, restoring layers by assembling files or chunks from storage system with deduplication incurs a considerable additional overhead on pull time.

Therefore, in this paper, we model the deduplication for estimating the deduplication effect on performance and storage savings, especially in terms of deduplication rate and deduplication overhead. We use Markov decision process to find optimal solution that can maximize the storage saving and minimizing the cost in terms of performance degradation.

2.Related work

MDPs are straightforward formalism decision-theoretic planning (DTP), reinforcement learning (RL); and other learning problems in stochastic domains. Markov environment can be viewed as a set of states and actions can be performed to control the system's state. The goal of MDP is to control the system in such a way that some performance criterium is maximized [5,6,7,8].

Typically, almost all the RL problems are modeled by using MDPs. MDPs are learning processes. It can deal with uncertainty, goals specified in terms of reward measures, and with changing situations. Moreover, it is aimed at solving the problem for every state, as opposed to a mere plan from one state to another [7,8].

We assume the Markov property is that the effects of an action taken in a state depend only on that state and not on the prior history[7]. MDPs consist of a set of possible states (S), a set of possible actions (A), transitions (T) between states and a reward function definition R(s, a).

Definition: A Markov decision process is a tuple (S, A, T, R) in which S is a finite set of states, A finite set of actions, T a transition function defined as T: S x A → Prob(S) and R a reward function defined as R : S × A × S → **R**. The model of MDP are determined by the transition function T and the reward function R together [7,8].

Policies: Given an MDP(S, A, T, R), a policy is a mapping from S to A. Formally, a deterministic policy π is a function defined as π : S → A. It is also possible to define a P stochastic policy as π : S × A → [0, 1] such that for each state s ∈ S, it holds that π(s, a) ≥ 0 and $\sum_{a \in A} \pi(s, a) = 1$
.

Application of a policy to an MDP is done in the following way.

1. Determine the current state s. For example, a start state s0 from the initial state distribution *I* is generated.
2. Execute action π(s). If the policy π suggest the action a0 = π(s0), this action is performed. Based on the transition function T and reward function R, a transition is made to state s1, with probability T (s0 , a, s1) and a reward r0 = R(s0 , a0 , s1) is received.
3. Goto step 1. MDP process continues, producing s0 , a0 , r0 , s1 , a1 , r1 , 2 , a2 , ...

The policy is part of the agent and its aim is to control the environment modeled as an MDP.

Objective functions:
The objective function maps infinite sequences of rewards to single real numbers. There are three solutions.
1. Set a finite horizon and just total the reward
2. Discounting to prefer earlier rewards
3. Average reward rate in the limit

Discounting is the most analytically tractable and most widely studied approach.

Markov Reward Process: A Markov Reward Process or an MRP is a Markov process with value judgment. An MRP is a tuple (S, P, R, γ) where S is a finite state space, P is the state transition probability function, R is a reward function where, Rs = $\mathbb{E}$[Rt+1 | St = S],
Which means how much immediate reward we expect to get from state S at the moment.

The value function Gt, which is the total discounted rewards from time step t. Its goal is to maximize Gt [7],

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

γ is a discount factor, which informs the agent of how much it should reward in the future.

3.Technical quality

3.1 Cost model and formulation of deduplication as a MDP task

(1) Estimating deduplication performance:

In the original non-deduplication docker registry backend storage system, a pull request is processed in three steps: namely, docker registry searching out the image, transferring it though network and decompressing the layer gzip compressed tar files.

Therefore, we can formulate the latency as follows:

$$\text{T\_pull} = (\text{T\_lookup} + \text{T\_store\_I/O}) + \text{T\_network} + \text{T\_decompression}$$

On the other hand, when we apply deduplication in docker registry backend storage system, the pull latency can be expressed as follows:

$$\text{T\_pull} = (\text{T\_lookup} + \text{T\_store\_I/O}) + \text{T\_network} + \text{T\_tar} + \text{T\_compression}$$

The deduplication latency is:

$$\text{T\_dedup} = \text{T\_decompression} + \text{T\_lookup} + (\text{T\_update\_TAB} + \text{T\_networking} + \text{T\_store\_I/O}) \times (1 - R\_dup)$$

It means that when a write is detected as duplicate, it pays the (T_decompression + T_searching). Otherwise, it pays the additional (T_update_TAB + T_networking + T_store)

(2)Reward function:

The deduplication task can be formulated as a continuing discounted MDP. The goal is to optimize the overall deduplication performance and pull performance. We define the reward function based on pull performance and deduplication performance. The state spaces are the resource utilization based on the layers that are deduplicating. Actions are the change to the resource utilization based on the number of layers that are chosen to be deduplicated.

The long-term cumulative reward is the optimization target of MDP. In the deduplication task, the desired resource utilization (or the set of layers that are about to deduplicated) are the ones which optimize system-wide performance with higher deduplication ratio. The immediate rewards are the summarized deduplication and pull performance feedbacks on the resulted new resource utilization. The deduplication performance is measured by a score which is the ratio of registry throughput with deduplication (thrpt_ded, thrpt_pull) to a reference throughput without deduplication(ref_thrpt_nondedup, ref_thrpt_pull) plus a ratio of space saved to total space plus possible penalties when response time (resp) based SLAs (Service Level Agreement) are violated:

$$penalty\_pull = \begin{cases} 0 & if\ pull\_resp \leq SLA\_pull \\ \frac{pull\_resp}{SLA\_pull} & if\ pull\_resp > SLA\_pull \end{cases}$$

$$penalty\_thrpt = \begin{cases} 0 & if\ push\_resp \leq SLA\_push \\ \frac{push\_resp}{SLA\_push} & if\ push\_resp > SLA\_push \end{cases}$$

$$score = (\frac{thrpt\_dedup}{ref\_thrpt\_nondedup} \times \gamma + \frac{thrpt\_pull}{ref\_thrpt\_pull} \times (1-\gamma)) \times \delta + \frac{space\_savings}{Total\_space} \times (1-\delta)$$

The reference throughput (ref_thrpt_nondedup) values are the maximum achievable throughput without deduplication in current docker registry settings. A low score indicates either lack of resource or high deduplication overhead, both of which should be avoided in making allocation decisions. Then, the immediate reward is the summarized scores over all layer deduplicated defined as:

$$\text{reward} = \begin{cases} \sqrt[n]{\coprod_{i=1}^{n} w_i \times score_i} & if\ for\ all\ score_i > 0 \\ -1 & otherwise, \end{cases}$$

(3) Optimal solution

The solution to a MDP task is an optimal policy that maximizes the cumulative rewards at each state. Here, we aim to find an estimation of Q(s, a) which approximates its actual value. We use the average of the sample Q(s, a) values collected to approximate the actual value of Q(s, a) given sufficiently large number of samples. We use temporal-difference (TD) methods, which update Q(s, a) at each time a sample is collected:

$$Q(s_t, a_t) = Q(s_t, a_t) + a \times [r_{t+1} + \gamma \times Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] ,$$

where α is the learning rate and γ is the discount factor.

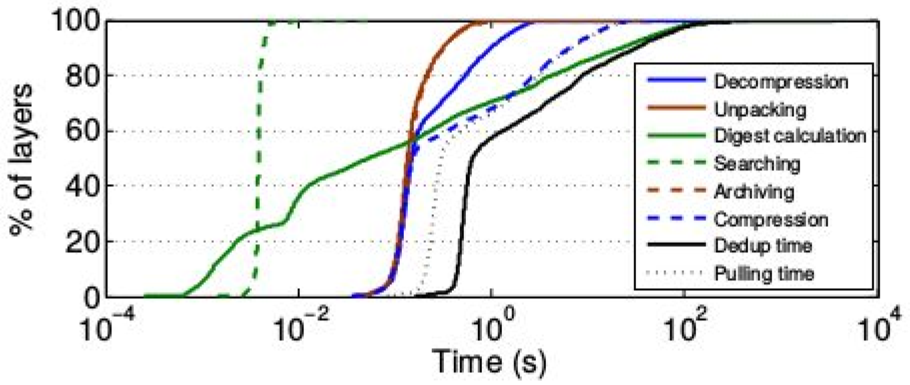

Figure 1 Performance analysis

3.2 simulation analysis

To analyze the impact of file-level deduplication on performance, we conduct a preliminary simulation-based study. First, a layer from our layer dataset is copied to a RAM disk. The layer is then decompressed, unpacked, and the fingerprints of all files are computed using the MD5 hash function. The simulation searches the fingerprint index for duplicates, and, if the file has not been stored previously, it records the file's fingerprint in the index. At this point our simulation does not include the latency of storing unique files. To simulate the layer reconstruction during a pull request, we archive and compress the corresponding files.

The simulator is implemented in 600 lines of Python code and our setup is a one-node Docker registry on a machine with 32 cores and 64 GB of RAM. To speed up the experiments and fit the required data in RAM we use 50% of all layers and exclude the ones larger than 50 MB. We process 60 layers in parallel using 60 threads. The entire simulation took 3.5 days to finish.

Figure 1 shows the CDF for each sub-operation of Slimmer. Unpacking, Decompression, Digest Calculation, and Searching are part of the deduplication process and together make up the Dedup time. Searching, Archiving, and Compression simulate the processing for a pull request and form the Pulling time.

4.Conclusion

From Figure 1 we can see that 55% of the layers have close compression and archiving times ranging from from 40 ms to 150 ms and both operations contribute equally to pulling latency. After that, the times diverge and compression times increase faster with an 90th percentile of 8 s. This is because compression times increase for larger layers and follow the distribution of layer sizes (see Figure 1). Compression time makes up the major portion of the pull latency and is a bottleneck. Overall, the average pull time is 2.3 s.

First, the searching time is the smallest among all operations with 90% of the searches completing in less than 4 ms and a median of 3.9 ms. Second, the calculation of digests spans

a wide range from 5 μs to almost 125 s. 90% of digest calculation times are less than 27 s while 50% are less than 0.05 s. The diversity in the timing is caused by a high variety of layer sizes both in terms of storage space and file counts. Third, the run time for decompression and unpacking follows an identical distribution for around 60% of the layers and is less than 150 ms. However, after that, the times diverge and decompression times increase faster compared to unpacking times. 90% of decompressions take less than
950 ms while 90% of packing time is less than 350ms. Overall, we see that 90% of file-level deduplication time is less than 35 s per layer, while the average processing time for a single layer is 13.5 s. This means that our single-node deployment can process about 4.4 layers/s on average (using 60 threads) based on MDP.

Reference

[1] P. Menage, “Adding Generic Process Containers to the Linux Kernel,” in Linux Symposium’07.
[2] 451 Research, “Application Containers Will Be a $2.7Bn Market by 2020,” https://tinyurl.com/ya358jbn.
[3] “Docker Hub,” https://hub.docker.com/.
[4] “GitHub,” https://github.com/.
[5] M. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming, ser. Wiley Series in Probability and Statistics. Wiley, 2014. [Online]. Available: https://books.google.com/books?id=VvBjBAAAQBAJ
[6] E. Altman, Constrained Markov Decision Processes, ser. Stochastic Modeling Series. Taylor & Francis, 1999. [Online]. Available: https://books.google.com/books?id=3X9S1NM2iOgC
[7]
https://towardsdatascience.com/reinforcement-learning-demystified-markov-decision-processes-part-1-bf00dda41690
[8] Martijn van Otterlo, Markov Decision Processes: Concepts and Algorithms.