On the Performance of MapReduce: A Stochastic Approach

Sarker Tanzir Ahmed and Dmitri Loguinov*
Texas A&M University, College Station, TX 77843, USA
Email: {tanzir, dmitri}@cse.tamu.edu

***Abstract*—MapReduce is a highly acclaimed programming paradigm for large-scale information processing. However, there is no accurate model in the literature that can precisely forecast its run-time and resource usage for a given workload. In this paper, we derive analytic models for shared-memory MapReduce computations, in which the run-time and disk I/O are expressed as functions of the workload properties, hardware configuration, and algorithms used. We then compare these models against trace-driven simulations using our high-performance MapReduce implementation.**

Keywords – MapReduce; Big Data; Disk I/O; External Sort

I. Introduction

MapReduce is a widely accepted programming model for large-scale data analytics. It has received much attention and adoption since its introduction by Google [2]. The motivation behind MapReduce is to provide a simple abstraction for large-scale information processing, which is a common problem in many enterprizes. To handle massive data sets, MapReduce programs first split them into smaller manageable fractions, compute partial solutions over them, and then write the results to secondary storage as *disk spill*s. Since the total computation time depends on the amount of disk spill, its accurate analysis is required for obtaining usable performance models of MapReduce.

There has been some limited work towards characterizing MapReduce performance [1], [5], [7], [15]. Their main limitation is that they assume a constant multiplicative factor that converts the size of input (intermediate data) to that of disk spill. In practice, however, the volume of disk I/O in a MapReduce computation depends non-linearly on the task, workload characteristics, used algorithms, and data structures. To the best of our knowledge, none of the existing work can fully quantify disk spill in terms of the above factors. The primary motivation of this work is to address this problem.

A MapReduce workload can be viewed as a multi-set of key-value pairs. Often, the MapReduce task is to combine the values with the same keys. Define $p(t)$ to be the probability that the t-th key is not seen before time t. In this paper, we first derive p(t) as a function of the frequency distribution of the keys and use this result to obtain an accurate disk-spill model of MapReduce. Equipped with these results, we then obtain a model for the total runtime of MapReduce programs, focusing primarily on the merge-sort design of shared-memory MapReduce, where a single host performs all MapReduce jobs. We finish by comparing the obtained models against simulation under different workloads and various resource constraints.

*Supported by NSF grants CNS-1017766 and CNS-1319984.

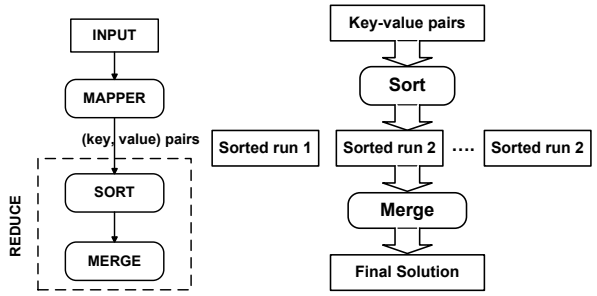
II. Related Work

Performance of MapReduce and its Hadoop implementation have been explored to some extent in previous work. The authors in [7] formulate a model for the disk I/O between the map and reduce phases, and also for the merge in the reduce phase. They consider both single-phase and multi-phase merge, which is required when the number of sorted files is too large. After that, the authors derive an analytic model for the job completion time as a function of configuration parameters of MapReduce (e.g., number of mappers, reducers), CPU and disk speed, and then use this model for optimizing various Hadoop parameters. They also present a number of hash-based techniques to improve the basic Hadoop implementation to achieve faster completion time. However, the authors assume a known constant multiplying factor between input and output size of various stages, which is not known beforehand and requires a separate training run for *each* value of RAM size.

In [1], the authors derive an optimal scheduling strategy of map and reduce jobs using Divisible Load Theory (DLT) with the goal of obtaining the shortest completion time. Similar to [7], they also assume a constant factor between input and output size. In [14], the authors focus on job deadlines and derive upper/lower bounds on computation time under the Earliest Deadline First (EDF) scheduling policy. But these theoretical bounds are loose in practice and hold only under EDF scheduling. The optimization methods proposed in [3], [12], [15] all require collection of previous job execution traces and then machine-learning approaches for determining optimal MapReduce/Hadoop parameters.

Phoenix [10] is the first MapReduce framework for shared memory rather than a distributed setting. The focus of this paper is to examine the effectiveness of the MapReduce framework in multi-core systems. The authors propose a grid-style data distribution strategy between the map and reduce workers. In [13], the authors examine the performance of Phoenix and Metis [8], another shared-memory MapReduce framework, and derive analytic models for both systems. The authors in this paper show that MapReduce performance depend on the ordering and frequency distribution of the keys, used data structures, and algorithms. However, they assume

978-1-4799-5666-1/14/$31.00 ©2014 IEEE



(a) MapReduce (b) reduce phase

Fig. 1: MapReduce with external sort.

the same frequency for all keys, which is unrealistic in many cases.

III. Approach

We start this section by stating the main goal of the paper. This is done by first discussing the general architecture of a MapReduce program. Then we move to the formulations and derivations of the basic metrics that are required for obtaining the actual performance models.

A. Objective

The MapReduce programming model consists of two phases – *map* and *reduce*. As shown in Fig. 1(a), the former processes the input data using a user-provided parsing function and produces a stream of key-value records called *intermediate data*. After the map phase finishes, these intermediate data are sorted by the key and then combined using a user-provided reduce function. Due to memory limitations, the reduce phase often has to process only a portion of the intermediate data and generate partial solutions, which we call *sorted runs*. These are then merged in the *merge* sub-phase to compute the final solution. Fig. 1(b) shows the reduce phase in detail.

Note that the sorted runs are written to disk during the sort phase and read back to RAM at least once during the merge phase. Thus, their combined volume, which we call *disk spill* I, plays an important role in determining the total runtime. In this work, the primary objective is to present an analytical model for the disk spill in shared-memory MapReduce computations.

B. Terminology

We consider both the input to a MapReduce program and its output as data streams consisting of key-value records. Assume a set V of unique keys, whose size is n. Define $\mathcal{I}(v)$ to be the frequency of v in the stream, and $T = \sum_{v \in V} \mathcal{I}(v)$ as the stream's length. Then, a realization of the stream can be viewed as some stochastic process $\{X_t\}_{t=1}^{T}$, where X_t is the random key, possibly accompanied by some value, in position $1 \leq t \leq T$. For simplicity of presentation, let random variable $\mathcal{I}$ have the same distribution as the frequency of the system:

$$P(\mathcal{I} < x) = \frac{1}{n} \sum_{v \in V} \mathbf{1}_{\mathcal{I}(v) < x}, \tag{1}$$

where we assume that the distribution of $\mathcal{I}$ is known a-priori. This is realistic when the same stream is queried multiple times for different purposes, or its properties can be predicted from other analysis. Note that RAM size R and sorting algorithm can be arbitrary. In prior work that uses empirical trace data, experiments must be repeated for each new configuration. Our results do not suffer from this drawback.

The output of the MapReduce program is a list of size n of key-value pairs, one for each $v \in V$. The above definition is general enough to describe many real-world data streams (e.g., user clicks, DNS queries, web page requests, random graphs) and a range of MapReduce tasks. The input data can directly follow the stream definition or result from some previous stage in a multi-stage MapReduce.

As the input stream is being processed, let $S_t = \bigcup_{i=1}^{t} X_i$ be the set of keys *seen* by time t. Furthermore, assume that set $U_t = V \setminus S_t$ contains the *unseen* keys at the same time. Now, the probability $p(t)$ that key X_t has not been seen before time t is defined as:

$$p(t) = P(X_t \in U_{t-1}), \tag{2}$$

which is the most important metric as it helps derive other quantities of interest. We next state the assumptions required to make this model tractable.

C. Assumptions

We assume that the $\mathcal{I}(v)$ copies of v are spread uniformly across the length of the stream, making the probability to encounter v at time t:

$$P(X_t = v) = \frac{\mathcal{I}(v)}{T}, \tag{3}$$

which is sometimes called the *Independent Reference Model* [9]. Thus, keys with high frequency/degree are more likely to be seen, irrespective of which portion of the stream is being examined.

An interesting conclusion entails from the assumption under the condition that v has not been seen before t. Given that v is unseen, there are still $\mathcal{I}(v)$ copies of v in the stream, where a total of $T - t + 1$ keys are yet to be processed. Therefore, the probability of seeing v at time t is the same as that of choosing any of $\mathcal{I}(v)$ items from $T - t + 1$ possibilities. Therefore,

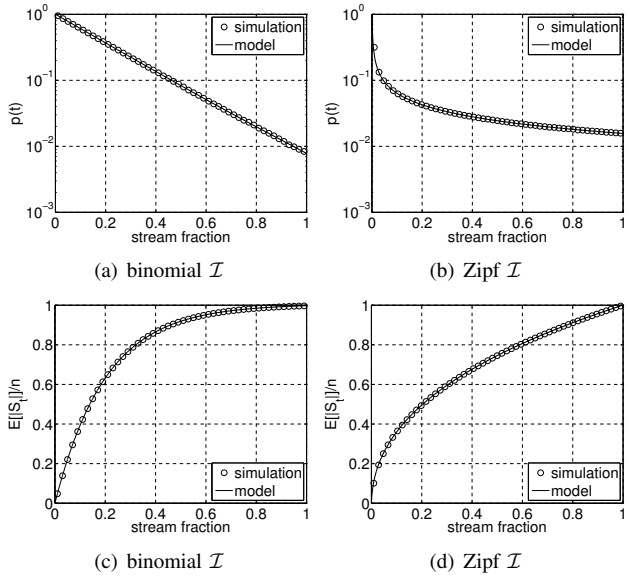
$$P(X_t = v | v \in U_{t-1}) = \frac{\mathcal{I}(v)}{T - t + 1}. \tag{4}$$

D. Formulation

We start with the likelihood for a given key v to remain unseen throughout the interval $[1, t]$. We use $\epsilon_t = t/T$ for notational convenience.

Theorem 1. *The probability that v is still unseen at time t:*

$$p(v, t) = P(v \in U_t) \approx (1 - \epsilon_t)^{\mathcal{I}(v)}. \tag{5}$$



(a) binomial $\mathcal{I}$ (b) Zipf $\mathcal{I}$

(c) binomial $\mathcal{I}$ (d) Zipf $\mathcal{I}$

Fig. 2: Comparison of models (8) and (11) to simulations.

Proof: For a node to stay unseen at t, it must be unseen at $t-1$ and also not hit at t. Therefore, using (4):

$$\begin{aligned} p(v,t) &= P(X_t \neq v, v \in U_{t-1}) \\ &= P(X_t \neq v | v \in U_{t-1}) P(v \in U_{t-1}) \\ &= \left(1 - \frac{\mathcal{I}(v)}{T-t+1}\right) p(v,t-1). \end{aligned} \tag{6}$$

After expanding the recurrence in (6) to $t=1$, we get:

$$p(v,t) = p(v,0) \prod_{\tau=0}^{t-1} \left(1 - \frac{\mathcal{I}(v)}{T-\tau}\right), \tag{7}$$

which produces (5) after using $p(v,0) = 1$ and Taylor approximation. ∎

Now, we are ready to derive the uniqueness probability which is the main result of this section.

Theorem 2. *The probability that the t-th key in the stream X_t refers to a previously-unseen node is:*

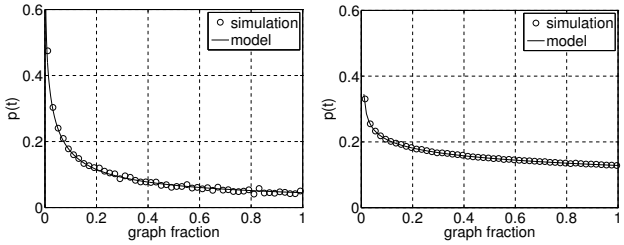
$$p(t) = P(X_t \in U_{t-1}) \approx \frac{E[\mathcal{I} \cdot (1-\epsilon_t)^{\mathcal{I}-1}]}{E[\mathcal{I}]}. \tag{8}$$

Proof: Partitioning the probability space into n mutually exclusive events, we get:

$$\begin{aligned} p(t) &= \sum_{v \in V} P(X_t \in U_{t-1}, X_t = v) \\ &= \sum_{v \in V} P(X_t = v | v \in U_{t-1}) P(v \in U_{t-1}). \end{aligned} \tag{9}$$

Using (4) and (5) in (9), we get:

$$\begin{aligned} p(t) &\approx \sum_{v \in V} \frac{\mathcal{I}(v)}{T-t+1} (1-\epsilon_t)^{\mathcal{I}(v)} \\ &\approx \frac{1}{T} \sum_{v \in V} \mathcal{I}(v) (1-\epsilon_t)^{\mathcal{I}(v)-1}. \end{aligned} \tag{10}$$



(a) IRLbot host graph (b) WebBase web graph

Fig. 3: Comparison of $p(t)$-model and simulation on graphs.

Since $\sum_{v \in V} f(\mathcal{I}(v)) = nE[f(\mathcal{I})]$ by our definition of $\mathcal{I}$, $T = nE[\mathcal{I}]$, and as before $(t-1)/T \approx \epsilon_t$, (10) produces (8). ∎

The results below follow immediately from Theorem 2.

Lemma 1. *The expected sizes of the seen and unseen sets at time t are respectively:*

$$E[|S_t|] = n - nE\left[(1-\epsilon_t)^{\mathcal{I}}\right], \tag{11}$$

$$E[|U_t|] = nE\left[(1-\epsilon_t)^{\mathcal{I}}\right]. \tag{12}$$

Proof: The size of the unseen set at time t is:

$$|U_t| = \sum_{v \in V} \mathbf{1}_{v \in U_t}, \tag{13}$$

which, after taking expectation on both sides becomes:

$$E[|U_t|] = \sum_{v \in V} P(v \in U_t) = \sum_{v \in V} (1-\epsilon_t)^{\mathcal{I}(v)} = nE\left[(1-\epsilon_t)^{\mathcal{I}}\right].$$

Then, $E[|S_t|]$ is given by $E[|S_t|] = n - E[|U_t|]$ which leads to (11). ∎

In Fig. 2(a) and 2(b), the $p(t)$ model in (8) is compared against simulation on binomial and Zipf ($\alpha = 1.5$) frequency distributions of the keys. It is apparent the model accurately captures the uniqueness probability in both cases. Note that the $p(t)$ behavior in the two cases is quite different. In the binomial case, $p(t)$ decreases exponentially. On the other hand, it drops more rapidly for the Zipf distribution in the beginning and then settles to a much slower (power-law) decay. In addition, Fig. 2(c) and 2(d) examine (11) in the same two scenarios and show this model to be accurate as well.

Next, we evaluate the $p(t)$ model on two real-world data sets – a host-level out-graph produced by IRLbot [6] and a URL out-graph of WebBase [11], both dating back to June 2007. The former contains 640M nodes, 6.8B edges and occupies 55GB of disk space. The latter contains 667M nodes, 4.2B edges and takes 32GB on disk. The graphs are represented using adjacency lists $(x_i, y_{i1}, y_{i2}, \ldots)$, where y_{ik} is the k-th out-neighbor of x_i and all node IDs are 64-bit hashes. In the file, neighbors $\{y_{ik}\}_k$ appear in numerically ascending order and so do source nodes $\{x_i\}_i$. We use these workloads throughout the rest of the paper for different experiments, where the edges are processed by sequentially scanning the graph.

Fig. 3 shows a comparison between model (11) and simulations of the uniqueness probability in the observed neighbor labels $\{y_{ik}\}$. These experiments demonstrate that (8) accurately predicts the observed $p(t)$ values. Armed with these results, we next model MapReduce overhead.

IV. MapReduce Analysis

We assume that the value field in each key-value pair is a scalar. As a result, pairs (v, a) and (v, b) are reduced using some combiner function $\theta(.)$ to a single pair $(v, \theta(a, b))$, where $\theta(a, b)$ remains a scalar. Suppose each key and value take K and D bytes, respectively. The input workload contains T key-value pairs and thus occupies $T(K + D)$ bytes. Let I denote the amount of disk spill from all the cycles and R, the size of RAM. The final output from the merge phase is a list consisting of n records, where there is one record for each distinct key. Since the disk spills are written in the sort phase and then read back in the merge phase, the total amount of I/O is:

$$W = (K + D)(T + n + 2I). \tag{14}$$

To verify the derived models below, we use a simple MapReduce task – computing the earliest time t at which each node v is first encountered among out-neighbors $\{y_{ik}\}$. This task requires maintaining a time-stamp during the scan of the graph for each seen node v. The combiner function simply computes the minimum of the values, i.e., $\theta(t_1, t_2) = \min(t_1, t_2)$. As an example, suppose a URL v contains three in-edges, which are seen by MapReduce at times $t = 5, 100, 105$. Then, the mapper emits pairs $(v, 5)$, $(v, 100)$ and $(v, 105)$. Now, the output of the MapReduce task is $(v, 5)$, which is obtained by combining values $5, 100, 105$. Note that the naive method of keeping all nodes with their minimum time-stamp in memory will not work when the graph is large.

In each cycle of a merge-sort MapReduce, the key-value pairs are read from the stream and copied to an array in RAM. When the array is full, it is sorted by the key, duplicates are removed via $\theta(\cdot)$, and the result is written to disk as a sorted run. This process continues for a number of cycles until the entire input is consumed. After that, all sorted runs are merged again using $\theta(\cdot)$ and finally the resulting pairs are saved to disk.

A. Disk I/O

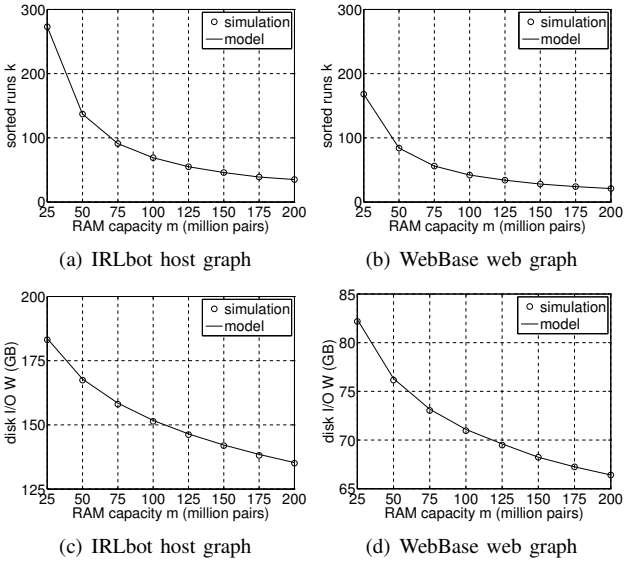
We start by computing the total disk I/O of a merge-sort-based MapReduce computation.

Theorem 3. *The amount of read/write I/O in a sort-merge MapReduce with a scalar combiner is:*

$$W = \frac{nR}{m}\left\{E[\mathcal{I}] + 1 + \frac{2}{\epsilon_m}\left(1 - E\left[(1-\epsilon_m)^{\mathcal{I}}\right]\right)\right\}, \tag{15}$$

where RAM capacity $m = R/(K + D)$.

Proof: Since each record takes $K + D$ bytes, the array in RAM of size R bytes can house $m = R/(K + D)$ pairs.



(a) IRLbot host graph (b) WebBase web graph

(c) IRLbot host graph (d) WebBase web graph

Fig. 4: Sorted runs and disk I/O in merge sort MapReduce.

Therefore, the number of sorted runs is:

$$k = \lceil \frac{T}{m} \rceil = \lceil \frac{T(K + D)}{R} \rceil. \tag{16}$$

Since each sorted run is effectively the seen set produced from a stream of length m, it contains:

$$E[|S_m|] = n - nE\left[(1-\epsilon_m)^{\mathcal{I}}\right], \tag{17}$$

pairs on the average. Hence, the total number of pairs in k sorted runs is:

$$\begin{aligned} I &= k \cdot E[|S_m|] = nk - nkE\left[(1-\epsilon_m)^{\mathcal{I}}\right] \\ &= \frac{n}{\epsilon_m}\left(1 - E\left[(1-\epsilon_m)^{\mathcal{I}}\right]\right), \end{aligned} \tag{18}$$

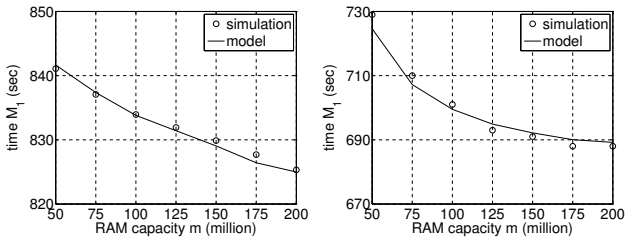
where the last step follows by using $k = T/m = 1/\epsilon_m$ instead of $\lceil T/m \rceil$ for simplicity. Then, we get (15) from (14). ■

Note in (15) that W is far from linear in input size T or RAM size R, which were assumed in existing literature [1], [5], [7], [15]. Fig. 4 compares models (15) and (16) against simulations in both graphs. It is evident that both models are accurate.

B. Sorting Time

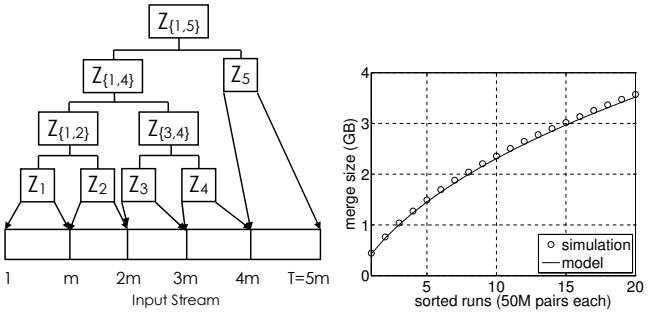
Using the above model for disk I/O, we now examine the total time for the sort/de-duplication phase. Assume that the average time for sorting m pairs is δ_m seconds, where $\delta_m = \Theta(m \log m)$. Furthermore, suppose the disk speed (both read and write) is ρ bytes/second. After sorting, only unique items in each array are written to disk after applying $\theta(\cdot)$. We call the delay of this operation *serialization time* and denote it by ξ_m. Note that this time is linear in m (i.e., $\xi_m = \mathcal{O}(m)$). Now, the total time for completing the sort and de-duplication is:

$$\mathcal{M}_1 = \frac{1}{\rho}(K + D)(T + I) + k(\delta_m + \xi_m). \tag{19}$$



(a) 500 MB/s disk (b) 2500 MB/s disk

Fig. 5: Total time (19) for the sort phase in Merge Sort MapReduce.



(a) selection tree (b) output pairs (23)

Fig. 6: Structure of the selection tree and merged output size.

In the above, the sorting time δ_m and ξ_m are determined for a particular m by just running one of the k cycles of the entire operation. All other parameters (e.g., ρ, m, K, D) are known due to the hardware configuration and the MapReduce task at hand.

To verify (19), we again compute the earliest time when each node in the IRLbot graph is seen. We vary the array capacity m from 50M to 200M pairs and each time run the entire sort/deduplication. We measure the completion time for two different disk read/write speeds ρ: 500 MB/s and 2500 MB/s. Fig. 5 shows a comparison between the model and measured times for the two configurations. The experiments demonstrate that the model closely predicts the runtime in both cases.

C. Merge Time

After the sort phase finishes, the merge phase combines all sorted runs. Let the merge speed be γ_m pairs per second. Therefore, the total time for this phase is:

$$\mathcal{M}_2 = \frac{1}{\rho}(K+D)(n+I) + \frac{I}{\gamma_m}, \tag{20}$$

where the first term is for disk I/O and the second for merging the sorted runs.

Now, the merge rate γ_m depends on the algorithms and data structures that are used for merging, and more importantly on m. In our design, we use a selection tree [4], which is known to roughly perform $\lceil \log_2 k \rceil$ comparison for each key in the input stream, where k is the number of sorted runs. But when merging multiple sorted runs from the same stream, many of them contain keys that are also present in other runs. Since the merge phase de-duplicates these repeated keys, the estimate $\lceil \log_2 k \rceil$ can be much different from the actual number of comparisons. In the following, we develop a more accurate model for the number of comparisons by taking de-duplications at each node of the tree into consideration. Unlike multi-pass merging, where groups of files are merged recursively due to memory constraints, here input is processes in one pass.

The structure of the selection tree is shown in Fig. 6(a), which is a strictly binary tree and there is one leaf node for every sorted run. Therefore, the 5-th sorted run in Fig. 6 is merged at level 2 instead of level 4. Let Z_i denote the i-th sorted run, which is the results of processing m pairs in the input stream. From (5), the probability for a key v to be present in Z_i is:

$$P\left(v \in Z_i\right) = 1 - p(v,m) = 1 - \left(1-\epsilon_m\right)^{\mathcal{I}(v)}. \tag{21}$$

Let $Z_{\{1,\tau\}}$ denote the set of pairs obtained after merging sorted runs $Z_1, Z_2, \ldots, Z_\tau$. Since each sorted run spans m pairs in the stream, τ sorted runs span $m\tau$ pairs. Therefore, the probability of v being present in any of the τ runs is:

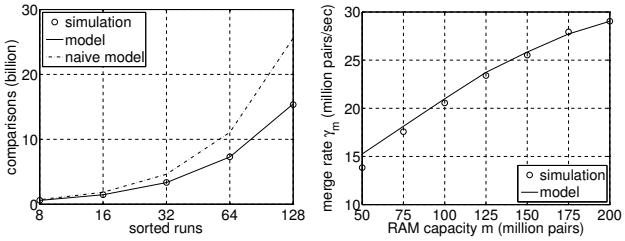
$$P\left(v \in \bigcup_{i=1}^{\tau} Z_i\right) = P(v \in Z_{\{1,\tau\}}) = 1 - \left(1-\epsilon_{m\tau}\right)^{\mathcal{I}(v)}, \tag{22}$$

which leads to the size of $Z_{\{1,\tau\}}$:

$$\begin{aligned} E\left[\left|Z_{\{1,\tau\}}\right|\right] &= \sum_{v \in V} \left(1 - \left(1-\epsilon_{m\tau}\right)^{\mathcal{I}(v)}\right) \\ &= n - nE\left[\left(1-\epsilon_{m\tau}\right)^{\mathcal{I}}\right]. \end{aligned} \tag{23}$$

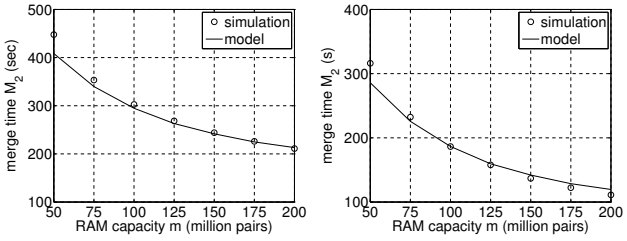
Note that the above relation holds not just for the sets $Z_1, \ldots, Z_\tau$, but also for any combination of τ sorted runs, as long as they span $m\tau$ pairs in the original stream. Fig. 6(b) compares model (23) against simulation on the IRLbot graph using RAM capacity $m = 50$M pairs, where we measure the number of key-value pairs in merged files obtained by combining a different number of randomly selected sorted runs. The comparison shows that the model is accurate. Experiments with other m values provide similarly accurate results.

Since the model in (23) enables computing the size of the sets $Z_{\{1,x\}}$ for any x, we can compute the number of comparisons performed in each internal node in the selection tree. As an example, the node labeled $Z_{\{1,2\}}$ in Fig. 6(a) does $E[|Z_1|] + E[|Z_2|] = 2E[|Z_1|]$ comparisons, but produces $E[|Z_{\{1,2\}}|]$ pairs for the parent node $Z_{\{1,4\}}$, which ultimately compares $2E[|Z_{\{1,2\}}|] = 4E[|Z_1|]$ pairs. Thus, for a complete binary selection tree with depth d that merges k sorted runs



(a) merge comparisons (b) merge speed γ_m

Fig. 7: Merge model and simulation.



(a) 500 MB/s disk (b) 2500 MB/s disk

Fig. 8: Total time (20) for the merge phase.

(i.e., $d = \log_2 k$), the total number of comparisons is:

$$C_k = \sum_{i=1}^{d} \frac{k}{2^{i-1}} E\big[|Z_{\{1,i\}}|\big] = nk \sum_{i=1}^{d} \frac{1}{2^{i-1}} \left(1 - E\big[(1-\epsilon_{im})^{\mathcal{I}}\big]\right). \quad (24)$$

On the other hand, the naive method that does not consider de-duplication of keys suggests the number of comparison to be same in each level. Thus, the number of comparisons in this case is:

$$\hat{C}_k = d \cdot I = dkE\big[|Z_1|\big]. \quad (25)$$

In Fig. 7(a), we compare model (24) and the naive model (25) against simulation on the IRLbot graph with RAM capacity $m = 50$M pairs. The experiment shows that as we have more sorted runs, the number of comparisons estimated by (25) increasingly deviates from the correct values. On the other hand, our model (24) predicts the actual number of comparisons very accurately.

In Fig. 7(b), we compare the derived merge rates γ_m against simulation on the IRLbot graph under various RAM capacities. We assume that the merge rate is inversely proportional to the total number of comparisons done in the selection tree. First, the entire graph is merge-sorted using $m = 200$M pairs, and the measured merge rate (without considering the time for disk I/O) is considered as the reference rate. Then, the merge rates for the other RAM capacities are computed from the corresponding selection tree structures and lengths of the sorted runs (24). Fig. 7(b) shows that the model is in close agreement with the observed results.

Fig. 8(a) shows the time for the merge phase when disk read and write speed are both 500 MB/s, while Fig. 8(b) shows the same with 2500 MB/s read/write speed. Both experiments show that the computed models are accurate. Notice that in both cases, the observed time is higher for smaller RAM capacities. Since there are many sorted runs in these cases (e.g., 137 runs for $m = 50$M), reading them from disk concurrently causes non-trivial latencies during disk seeking, leading to longer runtime than captured in the model.

V. Conclusion

In this paper, we proposed analytical models for a commonly used merge-sort MapReduce and established that performance metrics (i.e., runtime and resource usage) depend non-linearly on workload properties and hardware configurations. These models are useful in forecasting performance without the help of expert knowledge and ad-hoc parameter choices. The models are based on the uniqueness probability (8), which in conjunction with other formulations have significance beyond MapReduce analysis. Our merge-rate model for selection trees is novel as well. In future, we plan to extend our analysis to multi-core and parallel MapReduce designs.

References

[1] J. Berlinska and M. Drozdowski, "Scheduling Divisible MapReduce Computations," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 450 – 459, Mar. 2011.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. USENIX OSDI*, Dec. 2004, pp. 137–150.

[3] H. Herodotou, "Hadoop performance models," *arXiv preprint arXiv:1106.0940*, 2011.

[4] J. Katajainen and T. A. Pasanen, "In-place Sorting with Fewer Moves," *Information Processing Letters*, vol. 70, no. 1, pp. 31 – 37, 1999.

[5] E. Krevat, T. Shiran, E. Anderson, J. Tucek, J. J. Wylie, and G. R. Ganger, "Applying Performance Models to Understand Data-Intensive Computing Efficiency," DTIC Document, Tech. Rep., 2010.

[6] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, "IRLbot: Scaling to 6 Billion Pages and Beyond," Texas A&M University, Tech. Rep. 2008-2-2, Feb. 2008. [Online]. Available: http://irl.cs.tamu.edu/publications/.

[7] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A Platform for Scalable One-Pass Analytics Using Mapreduce," in *Proc. ACM SIGMOD*, Jun. 2011, pp. 985–996.

[8] Y. Mao, R. Morris, and M. F. Kaashoek, "Optimizing MapReduce for Multicore Architectures," *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*, 2010.

[9] J. McCabe, "On Serial Files with Relocatable Records," *Operations Research*, vol. 13, pp. 609–618, 1965.

[10] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *Proc. IEEE HPCA*, 2007, pp. 13–24.

[11] The Stanford WebBase Project. [Online]. Available: http://dbpubs.stanford.edu:8091/~testbed/doc2/WebBase/.

[12] F. Tian and K. Chen, "Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds," in *Proc. IEEE CLOUD*, 2011, pp. 155–162.

[13] D. Tiwari and D. Solihin, "Modeling and Analyzing Key Performance Factors of Shared Memory MapReduce," in *Proc. IEEE IPDPS*, 2012, pp. 1306–1317.

[14] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments," in *Proc. ACM ICAC*, 2011, pp. 235–244.

[15] X. Yang and J. Sun, "An Analytical Performance Model of MapReduce," in *Proc. IEEE CCIS*, Sep. 2011, pp. 306–310.