Modeling Randomized Data Streams in Caching, Data Processing, and Crawling Applications

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

Abstract—**Many BigData applications (e.g., MapReduce, web caching, search in large graphs) process streams of random key-value records that follow highly skewed frequency distributions. In this work, we first develop stochastic models for the probability to encounter unique keys during exploration of such streams and their growth rate over time. We then apply these models to the analysis of LRU caching, MapReduce overhead, and various crawl properties (e.g., node-degree bias, frontier size) in random graphs.**

I. Introduction

Large-scale distributed systems (e.g., Google, Facebook) operate on *streams* of key-value pairs that arise from disk/network-based processing of massive amounts of data. Due to the enormous size of input, streaming is the commonly used computational model, where the arriving items are scanned sequentially and processed one at a time. While MapReduce [6] is one application that by design employs streaming, other types of jobs can be modeled under the same umbrella. For example, caching can be viewed as computation on streams, where the frequency of duplicate items determines performance (i.e., hit rate). Large-scale graph-processing algorithms that output edges/nodes in bulk (e.g., BFS search) can also be reduced to streams, where performance may be determined by the size of the frontier (i.e., pending nodes), bias in the observed degree, and/or discovery rate of new vertices.

It is common to replace keys with their hashes and apply computation that outputs data in random order, either by design (e.g., reversing edges in graphs) or as byproduct of some previous computation (e.g., sorting by a different key in an earlier stage of MapReduce). This results in real workload consisting of *randomized* streams, in which keys are shuffled in some arbitrary order. Understanding statistical properties of these streams is an important area of research as it leads to better characterization of MapReduce, caching, graph exploration, and more general streaming. However, existing analysis is not just scattered across many fields [1], [2], [4], [5], [8], [9], [10], [12], [17], [21], but is also lacking in its ability to accurately model the stochastic properties of random streams.

In this work, we first formalize one-dimensional (1D) streams as discrete-time processes $\{Y_t\}_{t\geq 1}$, where each item Y_t observed at time t is unique (i.e., previously unseen) with some probability $p(t)$ and duplicate otherwise. Given the frequency distribution of keys, we first derive $p(t)$ and obtain the number of unique keys observed by time t. Then, we extend our modeling framework to two-dimensional (2D) streams $\{X_t, Y_t\}_{t\geq 1}$, where random variables X_t and Y_t are correlated due to the nature of the workload (e.g., adjacent nodes in a graph).

To demonstrate the usefulness of the derived results, we apply them to obtain the overall miss rate of Least Recently Used (LRU) caches, with results verified over both simulated and real streams. Next, we analyze MapReduce computation, where we obtain the amount of data generated by each sorted run as a function of RAM size and, consequently, the total I/O overhead of the reduce phase. This has recently emerged as an important problem [2], [9], [21], with no prior closed-form analysis. Finally, we apply our 2D stream models to characterize Breadth First Search (BFS) crawls on directed random graphs and develop a number of interesting results (e.g., degree distribution of seen/unseen/crawled nodes, frontier size), all functions of crawl time t. We also use these results to present a comparative analysis between BFS and a number of other crawling methods.

II. Literature Review

Although we examine a number of problems (i.e., LRU performance, MapReduce overhead and crawl characteristics) under one single framework of data streams, the existing literature on these problems is spread across multiple fields.

Caching: There has been much work [4], [5], [7], [8] on deriving the cache miss rate under various replacement policies (e.g., LRU, LFU, random, FIFO) and *specific* distributions of key frequency. For example, [8] provides an asymptotic analysis of LRU for Zipf and Weibull cases. Another set of results [4] studies the miss rate of *fixed* items in the dataset. In contrast, our LRU model applies to *all* distributions of key frequency and *random*, rather than fixed, keys in the stream.

MapReduce: Since their introduction, both MapReduce [6] and its open-source implementation Hadoop [20] have received a great deal of academic and industry attention. The authors in [2], [9], [13] offer models for the size of sorted runs produced from MapReduce computation and those in [13] extend these results to multi-pass merging. They assume a constant multiplicative factor that converts the size of input to the number of keys in the sorted runs; however, the complex dependency on RAM size, hidden in this relationship, is not modeled. We overcome this limitation by showing how to use our stream formalisms to combine RAM utilization with

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

statistical stream properties to obtain the exact size of sorted runs and total disk I/O.

Random Graphs: Crawl characteristics of BFS and its sampling bias have been studied extensively in the literature [1], [10], [12], [17]. A characterization of degree bias as a function of crawl fraction and a method for its correction are presented in [10]. This method's main limitation is that it considers only undirected random graphs. Furthermore, this approach requires a modified BFS where every edge is constructed at crawl time by selecting both ends randomly. Such modification fails to preserve source-destination edge pairing inherent in graph exploration. Instead, our analysis deals with directed graphs and does not require any modification to the BFS algorithm.

III. Randomized 1D Streams

We start by introducing our terminology, assumptions, and objectives. We then develop a stochastic model for the uniqueness probability $p(t)$ and the size of the unique set.

A. Terminology

Assume a collection V of n unique keys. Suppose $\mathcal{I}(v)$ for each $v \in V$ is the number of times v appears in the input stream. Using graph-theory terminology, we often call v a *node* and $\mathcal{I}(v)$ its *in-degree*. Define $T = \sum_{v\in V} \mathcal{I}(v)$ to be the length of the stream, which we assume is randomly shuffled. Then, realizations of the stream can be viewed as a 1-dimensional stochastic process $\{Y_1, \dots, Y_T\}$, where Y_t is the random key in position $1 \le t \le T$. For simplicity of presentation, let random variable $\mathcal{I}$ have the same distribution as the in-degree of the system:

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

As the stream is being processed, let $S_t = \bigcup_{i=1}^{t} \{Y_i\}$ be the set of keys *seen* by time t and suppose $U_t = V \setminus S_t$ contains the *unseen* keys at t. Then, define $p(t) = P(Y_t \in U_{t-1})$ to be the probability that key Y_t has not been seen before time t, which is the central metric for assessing performance of streaming algorithms. This includes crawl characterization, cache analysis, and MapReduce computation. To develop a tractable model for the uniqueness probability $p(t)$, we must place certain constrains on the appearance of keys in the stream, as discussed next.

B. Stream Residuals

We start by defining a special class of streams that commonly occur in practice (e.g., MapReduce data processing, DNS queries, workload arrival to web servers), where the keys are hashed to random values before being streamed from storage into the program and there is no (or little) correlation between adjacent items. Define $Z(v,t)$ to be an indicator variable of v being in position t, i.e.,

$$Z(v,t) = \begin{cases} 1 & Y_t = v \\ 0 & \text{otherwise} \end{cases}, \tag{2}$$

and consider the following.

Definition 1: A stream is called to possess *Uniform Residuals* (UR) if the probability of seeing v at time t is proportional to the number of remaining copies of v in the stream:

$$P(Y_t = v | Y_{t-1}, \dots, Y_1) = \frac{\mathcal{I}(v) - \sum_{\tau=1}^{t-1} Z(v,\tau)}{T - t + 1}. \tag{3}$$

Note that $\{Y_t\}_{t\ge1}$ is not a Markov chain since Y_t at every step depends on the entire history of the process. To understand (3) better, define $H(v,t)$ to be the number of times v is seen in $[1,t]$:

$$H(v,t) = \sum_{\tau=1}^{t} Z(v,\tau), \qquad t = 1, 2, \dots, T, \tag{4}$$

where $H(v,0) = 0$ and $H(v,T) = \mathcal{I}(v)$. While $H(v,t)$ is a sum of Bernoulli random variables, it is tempting to speculate that it is binomial; however, this is false since each $Z(v,t)$ depends on prior values $Z(v,1), \dots, Z(v,t-1)$ and:

$$\sum_{\tau=1}^{t} Z(v,\tau) \le \mathcal{I}(v), \tag{5}$$

which makes set $\{Z(v,t)\}_t$ non-iid.

Now, letting $R(v,t) = \mathcal{I}(v) - H(v,t)$ be the *residual degree* of v and unconditioning (3) by taking expectation over all sample paths, we get a more intuitive definition of UR streams:

$$P(Y_t = v) = \frac{\mathcal{I}(v) - E[H(v,t-1)]}{T-t+1} = \frac{E[R(v,t-1)]}{T-t+1}, \tag{6}$$

which shows that the probability to encounter v is proportional to its expected number of residual copies in the interval $[t,T]$. The rest of the analysis assumes that streams under consideration exhibit the UR property.

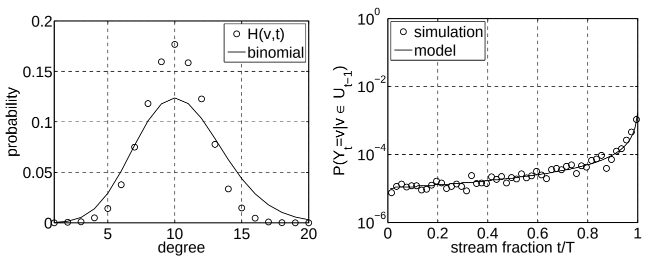
One interesting conclusion emerges from (6). Observe that the residual degree of an unseen key v always equals its total degree $\mathcal{I}(v)$. Therefore, the probability of discovering such nodes is:

$$P(Y_t = v | v \in U_{t-1}) = \frac{\mathcal{I}(v)}{T-t+1}. \tag{7}$$

C. A Few Words on Simulations

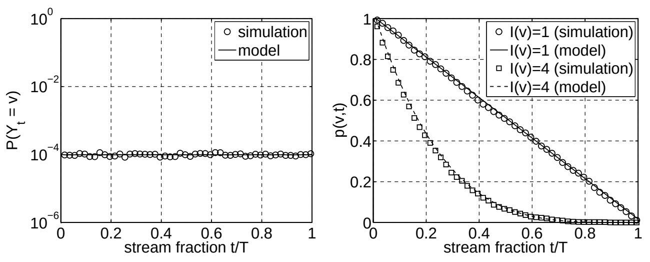
Throughout this section, we simulate MapReduce streams by first establishing sequence $\{\mathcal{I}(v)\}_{v\in V}$ under a given distribution. We use Zipf $\mathcal{I}$ with different shapes α as an approximation to in-degree of the web [3] and binomial as a model of degree in $G(n,p)$ random graphs. Then, we repeat each key v exactly $\mathcal{I}(v)$ times, assign it a random hash based on its position in the stream, and then sort the result by the hash, which gives us one realization of the stream. Changing the seed to the hash function, we execute this process a number of times to generate many sample paths of the system. In the next section, we use streams produced by crawls over random graphs and focus on a single sample-path. Finally, the last section of the paper employs real (non-simulated) input.

To put this discussion to use, Fig. 1(a) illustrates the distribution of $H(v,t)$ using a node with $\mathcal{I}(v) = 20$ and $t/T = 0.5$. Notice that the binomial fit has a much higher variance than the



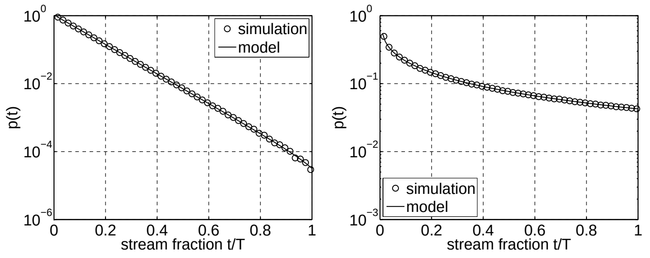
(a) distribution of $H(v,t)$ (b) model (7)

Fig. 1. Simulations with Zipf $\mathcal{I}$ under $\alpha = 1.2, E[\mathcal{I}] = 10, n = 10$K.



(a) model (10) (b) model (11)

Fig. 2. Verification under Zipf $\mathcal{I}$ with $\alpha = 1.2, E[\mathcal{I}] = 10, n = 10$K.



(a) binomial $\mathcal{I}$ (b) Zipf $\mathcal{I}$ ($\alpha = 1.2$)

Fig. 3. Verification of (13) under $E[\mathcal{I}] = 10, n = 10$K.

true distribution, which is a consequence of the dependency in (5). Fig. 1(b) verifies that (7) is indeed applicable to randomly sorted streams, where we track a random node with $\mathcal{I}(v) = 1$.

D. Single Node

Our examination begins with properties of a fixed node v, which we will need later for the more advanced results. Define *stream fraction* $\epsilon_t = t/T$ and consider the following result.

Lemma 1: Assume t and x are integers such that $0 \le x \le t \le T$. Then, the following holds:

$$\prod_{\tau=0}^{t-1}\left(1-\frac{x}{T-\tau}\right) = \prod_{\tau=0}^{x-1}\left(1-\frac{t}{T-\tau}\right) \approx (1-\epsilon_t)^x. \quad (8)$$

In our application of (8), t usually represents time and x the random degree of a node, where $x \ll T$ holds. Under these conditions, Lemma 1 is important in its ability to create a simple, yet very accurate approximation to a product of millions (if not billions) of terms. In fact, when $x = 1$, the $(1-\epsilon_t)^x$ term in (8) is exact. Leveraging this observation, we obtain the following result.

Theorem 1: The expected residual degree of v at time t is:

$$E[R(v,t)] = (1-\epsilon_t)\mathcal{I}(v). \quad (9)$$

Substituting (9) into (6), Theorem 1 shows that the probability of v being hit at t is time-invariant:

$$P(Y_t = v) = E[Z(v,t)] = \frac{\mathcal{I}(v)}{T}, \quad (10)$$

which is sometimes called the *Independent Reference Model* [15]. This result is confirmed in Fig. 2(a), which shows that $P(Y_t = v)$ indeed stays constant throughout the stream and its value is a function of $\mathcal{I}(v)$, but not the time. Thus, nodes with high degree are more likely to be seen irrespective of which portion of the stream is being examined. Interestingly, (10) shows that the distribution of $Z(v,t)$ does not depend on t and thus $\{Z(v,t)\}_t$ is a set of *identically-distributed* random variables; however, from (5), we know they are dependent.

The next result derives the likelihood for a given node v to remain unseen throughout an entire interval $[1,t]$.

Theorem 2: The probability that v is still unseen at time t:

$$p(v,t) = P(v \in U_t) \approx (1-\epsilon_t)^{\mathcal{I}(v)}. \quad (11)$$

To verify (11), we use a Zipf stream and select two random keys: one with degree 1 and the other with degree 4. Then, we observe the corresponding values of $p(v,t)$ in simulations and compare them with model (11) in Fig. 2(b), which shows that the model is accurate. As expected from (11), $p(v,t)$ is linear for the degree-1 node and nonlinear for the degree-4 node.

Armed with Theorem 2, we can solve the opposite problem from that in (7) to produce a useful result that will help us later establish how the degree of newly discovered nodes varies with time.

Theorem 3: Conditioned on the fact that node v is hit at time t, the probability that v was unseen at $t-1$ is:

$$P(v \in U_{t-1}|Y_t = v) \approx (1-\epsilon_t)^{\mathcal{I}(v)-1}. \quad (12)$$

E. Uniqueness Probability

We now are ready to obtain the main result of this section.

Theorem 4: The probability that the t-th key in the stream Y_t refers to a previously-unseen node is:

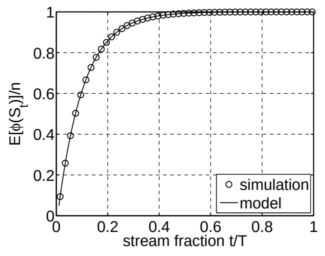
$$p(t) = P(Y_t \in U_{t-1}) \approx \frac{E[\mathcal{I}\cdot(1-\epsilon_t)^{\mathcal{I}-1}]}{E[\mathcal{I}]}. \quad (13)$$

To perform a self-check, we analyze $p(t)$ for two special cases. First, assume constant degree $\mathcal{I}(v) = d \ge 1$ for all $v \in V$. In this case, (13) simplifies to:

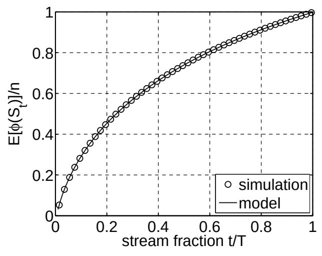
$$p(t) \approx (1-\epsilon_t)^{d-1}, \quad (14)$$

which is the probability that none of the Y_t's remaining $d-1$ appearances have fallen into the interval $[1,t)$. Second, consider a stream of *unique* items, i.e., $\mathcal{I}(v) = 1$ for all $v \in V$. In this scenario, (13) produces the correct $p(t) = 1$ for all t.

We compare Theorem 4 against simulations in Fig. 3. It is clear that in both cases the model is accurate in the entire range $[1,T]$. Also observe that $p(t)$ in the Zipf case (b) demonstrates

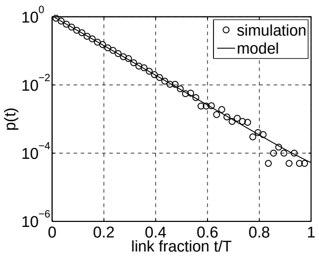


(a) binomial $\mathcal{I}$

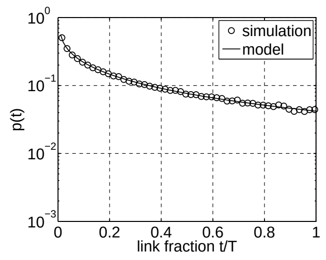


(b) Zipf $\mathcal{I}$ ($\alpha = 1.2$)

Fig. 4. Verification of (16) under $E[\mathcal{I}] = 10, n = 10\text{K}$.



(a) binomial $\mathcal{I}$



(b) Zipf $\mathcal{I}$ ($\alpha = 1.5$)

Fig. 5. Verification of $p(t)$ in BFS crawls with $E[\mathcal{I}] = 10, n = 10\text{K}$.

a heavier tail, which is a consequence of many low-degree nodes that stay undiscovered until the very end. On the other hand, the binomial curve quickly finds the majority of the nodes and $p(t)$ decays to zero exponentially fast. To explain this intuition better, we next analyze $p(t)$ as $t \to T$. Rewriting (13) by splitting between degree-1 nodes and all others:

$$p(t) \approx \frac{P(\mathcal{I} = 1) + E[\mathcal{I} \cdot (1 - \epsilon_t)^{\mathcal{I}-1} | \mathcal{I} > 1] P(\mathcal{I} > 1)}{E[\mathcal{I}]},$$

where the second term decays to 0 as $\epsilon_t \to 1$ and we get:

$$\lim_{t \to T} p(t) \approx \frac{P(\mathcal{I} = 1)}{E[\mathcal{I}]}, \tag{15}$$

which means that the last point in the curve is solely determined by the number of degree-1 nodes in the stream. Since $E[\mathcal{I}] = 10$ and nearly 40% of the nodes in the Zipf stream are degree-1, we immediately obtain $p(T) = 0.04$, which agrees with Fig. 3(b). On the other hand, the binomial stream contains fewer than 0.1% unique nodes and thus exhibits $p(T) < 10^{-4}$. Our experiments with a number of other in-degree distributions (e.g., uniform) and graph structures (e.g., hypercube) provide similarly accurate $p(t)$ results.

F. Set of Unique Nodes

Modeling the size of S_t allows characterization of cache performance and estimation of hash-table sizes required to store unique items as the stream is being processed. We come back to these issues later, but in the meantime define $\phi(A)$ to be the size of set A and present the following result.

Theorem 5: The expected size of the seen set at time t is:

$$E[\phi(S_t)] \approx nE\left[1 - (1 - \epsilon_t)^{\mathcal{I}}\right]. \tag{16}$$

We compare this result against simulations in Fig. 4 and observe that it is accurate. Note that the seen set grows more rapidly in the binomial case (a). For example, it accumulates 84% of the keys in the first 20% of the stream, while Zipf in subfigure (b) discovers only 45% of the nodes by that time. The reason is that $\phi(S_t)$ is essentially the integral of $p(t)$, which in the binomial stream contains much more mass in the beginning.

IV. Randomized 2D Streams

This section formalizes 2D streams and uses them for analyzing graph traversal algorithms (e.g., BFS, DFS).

A. Terminology and Assumptions

We start by describing a family of graphs where assumption (3) holds for common search algorithms. Consider a simple directed random graph $G(V, E)$, where V is the set of n nodes and E the set of T edges. To ensure uniform residuals, we generate in/out-degree sequences $\{\mathcal{I}(v)\}_{v \in V}$ and $\{\mathcal{O}(v)\}_{v \in V}$ according to the corresponding distributions such that:

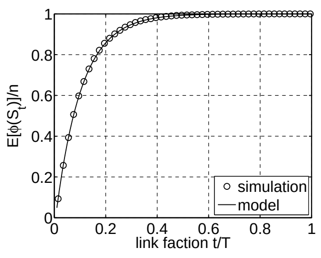
$$T = |E| = \sum_{v \in V} \mathcal{I}(v) = \sum_{v \in V} \mathcal{O}(v), \tag{17}$$

and then apply the so-called *configuration model* [16] to create random edges in the system. We describe this process next.

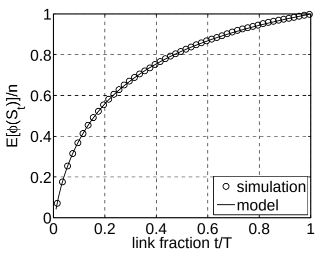
We use time-varying multi-sets $\mathcal{D}_{in}(t)$ and $\mathcal{D}_{out}(t)$ in the construction process. For each $v \in V$, we insert $\mathcal{I}(v)$ copies of the node into initial set $\mathcal{D}_{in}(0)$ and $\mathcal{O}(v)$ instances of it into $\mathcal{D}_{out}(0)$. At every step $1 \le t \le T$, a directed edge is formed between randomly selected nodes $x \in \mathcal{D}_{out}(t-1)$ and $y \in \mathcal{D}_{in}(t-1)$, both of which are then removed from the corresponding sets. Note that the generated graph G is a random instance among all possible graphs that can be constructed from a given degree sequence.

Next, we describe the crawl process. We consider algorithms that crawl/visit each node exactly once. Suppose at time t, node u is removed from the frontier F_t according to some exploration strategy. Then, all $\mathcal{O}(u)$ out-edges of node u, i.e., $(u, v_1), (u, v_2), \ldots, (u, v_{\mathcal{O}(u)})$, are processed at *consecutive* time steps $t, t+1 \ldots t + \mathcal{O}(u) - 1$, which creates correlation between the visited edges (i.e., they all have the same source node). If neighbor v_i is previously unseen, it is appended to the frontier F_{t+i-1}; otherwise, it is discarded. The crawler selects the next node from the frontier at $t + \mathcal{O}(u)$ and the process repeats.

We can now define the stream of edges produced by graph crawling as a 2D discrete-time stochastic process $\{(X_t, Y_t)\}_{t=1}^{T}$ on E, where X_t is the crawled node and Y_t the destination node (i.e., one of X_t's out-neighbors), both random variables. Randomness arises due to the stochastic nature of G, where each sample path $\{(X_t, Y_t)\}_{t=1}^{T}$ operates on a different instance of the graph. Defining $C_t = \bigcup_{i=1}^{t} \{X_i\}$ to be the set of already-crawled nodes by time t, frontier F_t can be expressed as $S_t \setminus C_t$. Before the crawl starts, both sets are empty, i.e., $C_0 = F_0 = \emptyset$. The choice of initial node X_0 does not affect the analysis and is omitted from the discussion.

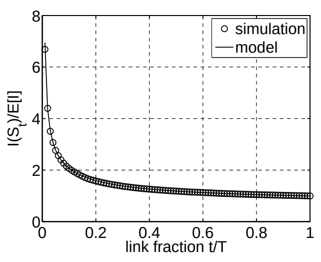


(a) binomial $\mathcal{I}$

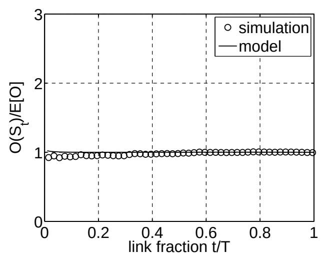


(b) Zipf $\mathcal{I}$ ($\alpha = 1.5$)

Fig. 6. Verification of $E[\phi(S_t)]$ in BFS crawls with $E[\mathcal{I}] = 10, n = 10\text{K}$.



(a) in-degree model (22)



(b) out-degree model (23)

Fig. 7. Verification with Zipf $\mathcal{I}$ under $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10\text{K}$.

B. Degree of the Seen Set

We start by examining the stream $\{Y_t\}_{t=1}^T$ of destination nodes produced by a crawler and verify that our earlier results are in fact applicable in this situation. Figs. 5-6 compare respectively models (13), (16) against BFS simulations on graphs with binomial and Zipf degree, where each graph contains the same number of nodes n and edges T. The figures show that both models are still very accurate. Armed with this confirmation, we draw our attention to inferring more advanced properties of the crawl process.

Let $\mathcal{I}(S_t)$ and $\mathcal{O}(S_t)$ denote respectively the number of in/out edges incident to the nodes in the seen set S_t, i.e.,

$$\mathcal{I}(S_t) = \sum_{v \in S_t} \mathcal{I}(v), \quad \mathcal{O}(S_t) = \sum_{v \in S_t} \mathcal{O}(v). \tag{18}$$

Then, we have the following result that helps understand the properties of the seen nodes and their average degree.

Theorem 6: The expected number of in/out edges incident to the nodes in the seen set is respectively:

$$E[\mathcal{I}(S_t)] \approx nE[\mathcal{I} \cdot (1 - (1-\epsilon_t)^{\mathcal{I}})], \tag{19}$$

$$E[\mathcal{O}(S_t)] \approx nE[\mathcal{O} \cdot (1 - (1-\epsilon_t)^{\mathcal{I}})], \tag{20}$$

where $\mathcal{O}$ is the random out-degree of a node in the system.

Interestingly, (19) can also be expressed as $p(t)T$, which shows that the in-degree of the seen set follows the same curve $p(t)$ scaled by the total number of edges T. The other model (20) captures correlation between in/out-degree in the system. If the two variables $\mathcal{I}$ and $\mathcal{O}$ are independent, the result expands to $nE[\mathcal{O}]E[1 - (1-\epsilon_t)^{\mathcal{I}}] = E[\mathcal{O}]E[\phi(S_t)]$. This makes sense as it multiplies the size of S_t by the average out-degree in the graph. However, when the two degree variables are dependent, the formula no longer admits a simple expansion.

With this general knowledge, we can quantify the degree bias of the nodes placed into the seen set. Define the average in/out-degree of the nodes in S_t respectively as:

$$\bar{\mathcal{I}}(S_t) = \frac{E[\mathcal{I}(S_t)]}{E[\phi(S_t)]}, \quad \bar{\mathcal{O}}(S_t) = \frac{E[\mathcal{I}(S_t)]}{E[\phi(S_t)]}, \tag{21}$$

and consider the following result.

Theorem 7: The average in/out-degree of the nodes in the seen set is:

$$\bar{\mathcal{I}}(S_t) \approx \frac{E[\mathcal{I} \cdot (1 - (1-\epsilon_t)^{\mathcal{I}})]}{1 - E[(1-\epsilon_t)^{\mathcal{I}}]}, \tag{22}$$

$$\bar{\mathcal{O}}(S_t) \approx \frac{E[\mathcal{O} \cdot (1 - (1-\epsilon_t)^{\mathcal{I}})]}{1 - E[(1-\epsilon_t)^{\mathcal{I}}]}. \tag{23}$$

In Fig. 7(a), we compare (22) against BFS simulations. The plot shows that the average in-degree in the seen-set starts at a much higher (i.e., 6 times in this example) value than $E[\mathcal{I}]$. Consistent with previous findings about BFS bias [1], [12], [17], the model confirms that BFS finds nodes with high in-degree earlier during the crawl. It also stochastically quantifies the amount of bias, which has not been shown before in the literature. To perform a self-check, notice that when $\mathcal{I}$ and $\mathcal{O}$ are independent, (23) simplifies to the average out-degree $E[\mathcal{O}]$, which is confirmed by Fig. 7(b).

C. Destination Nodes

Another related property is the degree distribution of destination nodes Y_t of the discovered edges as the crawl progresses.

Theorem 8: The in-degree distribution of Y_t is given by:

$$P(\mathcal{I}(Y_t) = k) = \frac{kP(\mathcal{I} = k)}{E[\mathcal{I}]}. \tag{24}$$

Note that (24) has been known in the literature [14] as the degree distribution of random walks on undirected graphs. The main benefit of this relationship is that bias-correction methods for random walks [10] can be applied to BFS in our scenarios. Specifically, using an observation of Y_t for $t \in [1, m]$, the following is an unbiased estimator of $P(\mathcal{I} = k)$:

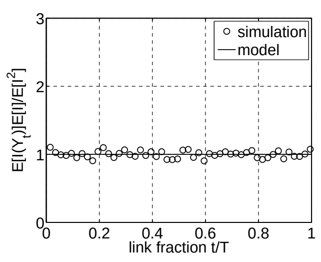
$$\frac{\sum_{t=1}^m \mathbf{1}_{\mathcal{I}(Y_t)=k}}{k\sum_{t=1}^m 1/\mathcal{I}(Y_t)}. \tag{25}$$

Next, we derive the expected degree of Y_t's.

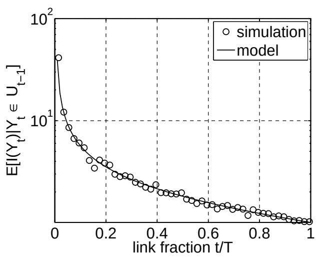
Theorem 9: The average in/out-degree of Y_t is independent of time and equals:

$$E[\mathcal{I}(Y_t)] = \frac{E[\mathcal{I}^2]}{E[\mathcal{I}]}, \qquad E[\mathcal{O}(Y_t)] = \frac{E[\mathcal{I}\mathcal{O}]}{E[\mathcal{I}]}. \tag{26}$$

We verify the in-degree model (26) against simulations in Fig. 8(a). Since we normalize both the model and the observed



(a) in-degree of Y_t (normalized)



(b) in-degree of unseen Y_t

Fig. 8. Verification of models (26) and (28) with Zipf $\mathcal{I}$ and $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10\text{K}$.

values by $E[\mathcal{I}^2]/E[I]$, the simulation staying constant at 1 indicates that the model is accurate.

The next lemma will become useful shortly in the analysis of nodes being added to the seen set at every time step t.

Lemma 2: The probability that the discovered node at t is equal to v, conditioned on its being unseen, is:

$$P(Y_t = v|Y_t \in U_{t-1}) \approx \frac{\mathcal{I}(v)(1-\epsilon_t)^{\mathcal{I}(v)-1}}{nE[\mathcal{I}(1-\epsilon_t)^{\mathcal{I}-1}]}. \qquad (27)$$

Now we are ready to derive the expected degree of the nodes that are moved from U_t into S_t as the crawl progresses. These are the nodes discovered for the first time at t.

Theorem 10: The expected in/out-degree of the discovered node at t, conditioned on its being unseen, is given by:

$$E[\mathcal{I}(Y_t)|Y_t \in U_{t-1}] = \frac{E[\mathcal{I}^2 \cdot (1-\epsilon_t)^{\mathcal{I}-1}]}{E[\mathcal{I}(1-\epsilon_t)^{\mathcal{I}-1}]}, \qquad (28)$$

$$E[\mathcal{O}(Y_t)|Y_t \in U_{t-1}] = \frac{E[\mathcal{I}\mathcal{O} \cdot (1-\epsilon_t)^{\mathcal{I}-1}]}{E[\mathcal{I}(1-\epsilon_t)^{\mathcal{I}-1}]}. \qquad (29)$$

We verify the in-degree model (28) against simulations in Fig. 8(b), which confirms the model. Interestingly, the expected degree of nodes added to S_t also starts at $E[\mathcal{I}^2]/E[\mathcal{I}]$ for $\epsilon_t = 0$ and then drops to $z = \min_{v\in V}\{\mathcal{I}(v)\}$ as $\epsilon_t \to 1$. To show the latter, we can apply L'Hôpital's rule to the ratio in (28) since both the numerator and denominator tend to zero. Thus, after some number of differentiations:

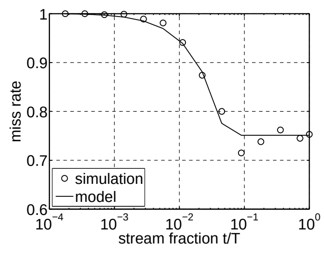
$$\lim_{\epsilon\to 1} \frac{\sum_{v\in V} \mathcal{I}(v)^2(1-\epsilon_1)^{\mathcal{I}(v)-1}}{\sum_{v\in V} \mathcal{I}(v)(1-\epsilon_1)^{\mathcal{I}(v)-1}} = \frac{z!zP(\mathcal{I}=z)}{z!P(\mathcal{I}=z)} = z. \quad (30)$$

However, unlike Fig. 7(a), which also begins at $E[\mathcal{I}^2]/E[\mathcal{I}]$, the decay rate of (28) is much faster.

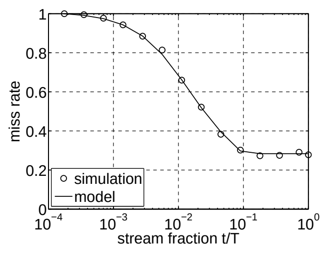
In summary, the models in this section characterize a crawl process by quantifying the various properties of Y_t to which the current edge (X_t, Y_t) points.

V. Applications

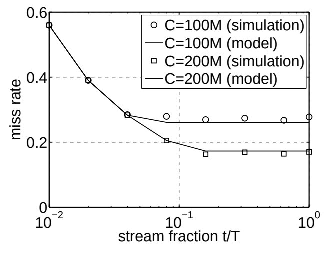
In this section, we examine a number of problems that deal with streams, where $p(t)$ is useful for measuring various quantities of interest. We first use it to analyze LRU cache performance, both on simulated and real streams. After that, we derive a stochastic model for the volume of data (also called *disk spill*) produced from MapReduce computation. Finally, we study the properties of X_t's in edge streams and obtain a number of metrics that apply to modeling crawls in random web graphs. Note that the first two cases work with 1D streams, while the last one employs 2D streams.



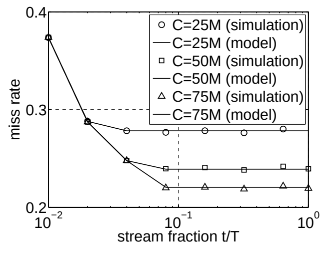
(a) random binomial stream



(b) random Zipf stream ($\alpha = 1.2$)



(c) IRLbot host graph



(d) WebBase web graph

Fig. 9. Verification of miss rate model (32).

In addition to simulated workloads, we use two real-world data sets for the experiments in this section. The first one is the host-level out-graph produced by IRLbot [11], while the second one is a URL out-graph of WebBase [19], both dating back to June 2007. The former graph contains 641M unique nodes and 6.8B links, standing at a hefty 60 GB. The second contains 635M unique nodes and 4.2B links. It is smaller at 35 GB, but still larger than RAM size of our servers, which requires disk-based streaming and batch-mode processing.

A. LRU Cache

We start by formulating the miss rate of an LRU cache using the $p(t)$ model (13). Assume a cache of capacity C data items, fed by an input stream of length T. Suppose that the cache is full (i.e., contains C unique keys) after processing τ keys from the stream, where τ is a function of C. Defining by $m(t)$ the miss rate of LRU at time t, we get:

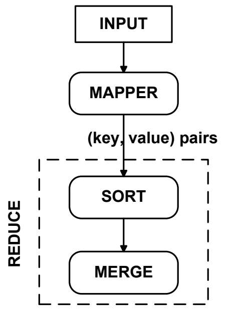
$$m(t) = \begin{cases} p(t) & t < \tau \\ p(\tau) & t \geq \tau \end{cases}. \qquad (31)$$

which can be explained by the fact that once the cache saturates (i.e., $t \geq \tau$), it experiences the same miss rate $p(\tau)$ for the rest of the stream. This can be simplified to:

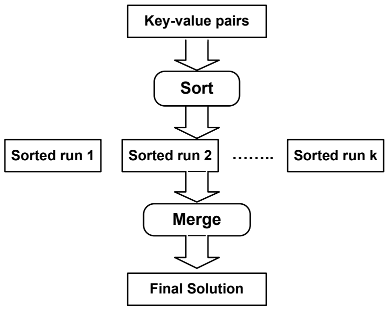
$$m(t) = p(\min(t,\tau))) = \frac{E[\mathcal{I} \cdot (1-\epsilon_{\min(t,\tau)})^{\mathcal{I}-1}]}{E[\mathcal{I}]}, \qquad (32)$$

To obtain τ, define $f(t)$ to be a monotonically increasing function equal to the expected number of unique items seen by time t:

$$f(t) = E[\phi(S_t)] = nE\left[1-(1-\epsilon_t)^{\mathcal{I}}\right]. \qquad (33)$$

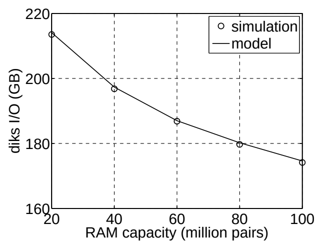


(a) MapReduce

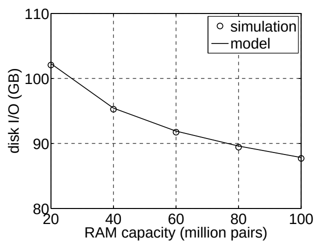


(b) reduce phase

Fig. 10. MapReduce with external sort.



(a) IRLbot host graph



(b) WebBase web graph

Fig. 11. Verification of (36).

Due to monotonicity of (33), the inverse $f^{-1}(.)$ exists and thus τ can be obtained as $f^{-1}(C)$. Although, there is no closed-form inversion to $f(t)$ unless $\mathcal{I}$ follows a well-known distribution, we can perform this job numerically. We compare model (32) against random streams in Fig. 9(a)-(b), where the x-axis is drawn on a log-scale. After the cache is saturated, the miss rate in (a) stays 3 times higher than in (b), which shows the ineffectiveness of caching in a binomial stream, where items are constantly evicted from the cache. On the other hand, the Zipf stream contains a few high-degree nodes and the LRU policy keeps most of them in the cache.

We test model (32) on the IRLbot graph, where we stream all edges sequentially from disk and pass them through an LRU cache. Fig. 9(c) shows the result with two cache sizes – 100M and 200M keys. The Webbase graph in Fig. 9(d) is run with $C =$ 25M, 50M, and 75M items. These figures show that the model is accurate for different types of input and cache sizes. Note that [4], which derives a popular LRU model, shows the miss rate for key v at time t as $e^{-\mathcal{I}(v)t/T}$, which can be viewed as Taylor expansion of the more accurate model for $p(v, t)$ in (11). In fact, comparing the two models, this approximation works only when $\mathcal{I}(v)t/T \approx 0$. Our work in this section is totally different from the results in [4], because we focus on the overall miss rate $m(t)$ rather than that of a fixed node v.

B. MapReduce

The MapReduce programming paradigm consists of two phases – *map* and *reduce*. In the former phase, input data are processed using a user-provided parsing function that outputs a stream of key-value pairs. It is then sorted and combined in the reduce phase, as illustrated in Fig. 10(a). Our work below analyzes the reduce phase whose details are given in Fig. 10(b). Specifically, the reduce phase begins by producing $k \geq 1$ *sorted runs*, each representing a portion of input that can fit in RAM. The sorted runs are then written to secondary storage after eliminating duplicates through some *combiner* function $\theta(.)$. We assume that key-value pairs (v, a) and (v, b) are combined into a single result $(v, \theta(a, b))$, where all keys and values are fixed-size scalars. After sorting is finished, the runs are streamed back to RAM and combined using k-way merge. The final output is another stream sorted by the key, in which each node v appears exactly once.

Assume that keys and values occupy K and D bytes, respectively. The main memory can store at most r key-value pairs and $r \ll T$. The MapReduce computation goes through $k = \lceil T/r \rceil$ cycles, each of which loads r records from the stream to memory, sorts the result, and eliminates duplicates by running the combiner. The size of each sorted run is given by the number of unique keys $q(r)$ out of the r loaded during the cycle. Unfortunately, there is no accurate model in the literature to compute this number. Most of the existing methods [2], [13] sidestep this problem by assuming some known constant that converts r into the size of each run $q(r)$.

Since the order of the keys are random, the streams under consideration fall under our assumptions in the beginning of the paper. This allows us to use model (16) to compute the size of each sorted run. Observe that in a stream of size r, the number of unique keys is given by model (16) as:

$$q(r) = E[\phi(S_r)] = nE[1 - (1 - \epsilon_r)^{\mathcal{I}}], \qquad (34)$$

which, unlike the assumption in [2], [9], [13], is far from linear in r. Since there are k sorted runs from k cycles, their total size is:

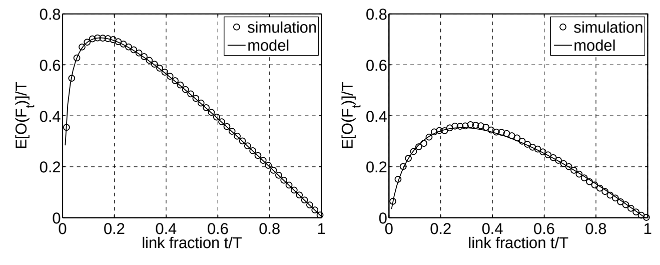
$$q = nk(K + D)E[1 - (1 - \epsilon_r)^{\mathcal{I}}]. \qquad (35)$$

Note that the full input stream of size $(K + D)T$ is read in the beginning and n unique pairs are written at the end. Counting both read and write I/O for each sorted run, we get the total amount of disk overhead as:

$$w = n(K + D)\left(1 + E[\mathcal{I}] + 2kE[1 - (1 - \epsilon_r)^{\mathcal{I}}]\right). \qquad (36)$$

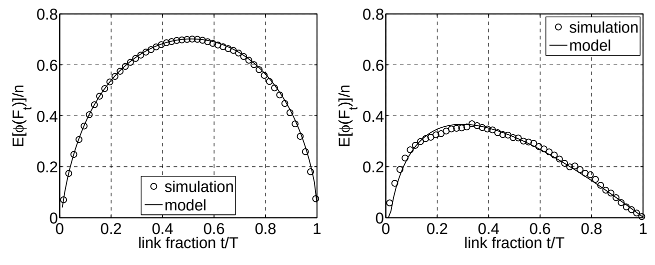
To examine (36), we use a MapReduce task that computes the earliest time $\delta(v) = \min\{t \geq 1 : Y_t = v\}$ each node v is seen in the stream. Therefore, each key-value pair consists of an 8-byte key (i.e., hash of the node ID) and an 8-byte timestamp representing the time of its earliest discovery. The combiner function is simply the minimum of the two values under consideration.

We run the above computation on the IRLbot and WebBase graphs under varying RAM capacities. The results are shown in Fig. 11, which demonstrates that the model is accurate in both cases. For example, Fig. 11(a) shows that the disk I/O for processing the IRLbot graph is 215 GB with $r =$ 20M pairs in RAM. This value of r represents 3.1% of n and



(a) correlated degree (b) uncorrelated degree

Fig. 12. Verification of frontier out-degree (40) with Zipf $\mathcal{I}$ and $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10$K.



(a) correlated degree (42) (b) uncorrelated degree (43)

Fig. 13. Verification of the frontier size in BFS with Zipf $\mathcal{I}$ and $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10$K.

0.29% of T. Excluding 70 GB of the input and final output, the remaining 145 GB are sorted runs. This demonstrates how intermediate I/O, necessary during disk-based sorting, constitutes a significant portion of MapReduce overhead.

Also observe that the results in (36) allow accurate modeling of the total completion time of MapReduce jobs (we omit the map phase as it is highly user-dependent and usually much faster than the reduce phase). Assume that the sorting and merging speed are respectively s and g keys per second. Furthermore, suppose the disk I/O speed is d bytes per second. Then, the total time needed for the reduce phase is:

$$\Gamma = \frac{T}{s} + \frac{w}{d} + \frac{nk}{g} E[1 - (1 - \epsilon_r)^{\mathcal{I}}]. \tag{37}$$

Result (37) specifies the MapReduce runtime for a given RAM size r. In addition to analyzing latency, this model can help optimize the MapReduce architecture by suggesting proper selection of r for a given (s, d, g), or vice versa.

C. Frontier in Graph Traversal

Frontier size is a crucial metric for crawlers, because it dictates the amount of resources required for maintaining crawl queues. The larger the crawl queue, the more overhead it places on uniqueness checks, which are needed to avoid re-crawling the same pages, and frontier prioritization, which sorts all pending nodes based on some importance metric. Here, we develop models of frontier size for a number of crawling strategies and present a comparison among them.

Recall that the frontier at time t is denoted by F_t. Similar to (18), define the combined out-degree of nodes in F_t as:

$$\mathcal{O}(F_t) = \sum_{v \in F_t} \mathcal{O}(v). \tag{38}$$

This can also be written as the number of out-edges emanating from the seen nodes minus the total out-degree of the nodes moved into the crawled set:

$$\mathcal{O}(F_t) = \sum_{v \in S_t} \mathcal{O}(v) - \sum_{v \in C_t} \mathcal{O}(v). \tag{39}$$

Taking an expectation of (39), we get:

$$E[\mathcal{O}(F_t)] = nE[\mathcal{O}(1 - (1 - \epsilon_t)^{\mathcal{I}})] - t. \tag{40}$$

We examine in simulations two cases of degree dependence: a) $\mathcal{O} = \mathcal{I}$ with a positive correlation; and b) $\mathcal{O}$ and $\mathcal{I}$ drawn independently from the same Zipf distribution. Fig. 12 compares model (40) against the observed values from BFS simulations, showing that the model is accurate. Observe that the curve in Fig. 12(a) is heavily skewed to the left, reaching its maximum value (i.e., $0.7T$) after seeing just 15% of the stream. On the other hand, $E[\mathcal{O}(F_t)]$ in Fig. 12(b) peaks after processing 30% of the stream, but at a significantly smaller value (i.e., $0.38T$). After the peaks, both curves drop to zero with a linear slope, just as predicted by the model. For exploration of large graphs of unknown size (e.g., the web), the shape of these curves provides insight into *crawl coverage* (i.e., percentage of links processed thus far) – positive slopes in (40) indicate that most links are yet to be seen.

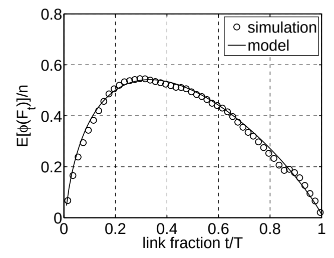
We finally arrive to the most interesting metric of this section – size of the frontier. Observe that $\phi(F_t)$ increases by addition of previously unseen nodes, which happens with probability $p(t)$, and removal of crawled nodes. Whenever a node X_t is removed from the frontier for crawling, all of its $\mathcal{O}(X_t)$ outgoing edges are processed before the next removal. Therefore, the rate of node removal from F_t is $1/\mathcal{O}(X_t)$. Approximating $E[1/\mathcal{O}(X_t)] \approx 1/E[\mathcal{O}(X_t)]$, the size of the frontier at t is:

$$E[\phi(F_t)] \approx E[\phi(F_{t-1})] + p(t-1) - \frac{1}{E[\mathcal{O}(X_{t-1})]}. \tag{41}$$

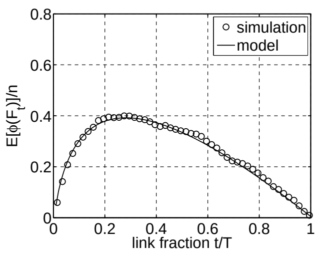
Quantity $E[\mathcal{O}(X_t)]$ depends on the crawl strategy that orders the frontier and the degree-properties of the graph. We next examine a number of such cases and compute the corresponding $E[\mathcal{O}(X_t)]$'s. First, we consider BFS with an arbitrary degree distribution. Note that after v is inserted into the BFS queue at t, all $\mathcal{O}(F_t)$ out-edges currently pending in F_t will be processed before v is crawled. Therefore, we can estimate the average time between the first discovery of a node at t and its crawl as $E[\mathcal{O}(F_t)]$. Using (29), this leads to:

$$\begin{aligned} E[\mathcal{O}(X_{t+E[\mathcal{O}(F_t)]})] &= E[\mathcal{O}(Y_t) | Y_t \in U_{t-1}] \\ &= \frac{E[\mathcal{I}\mathcal{O}(1 - \epsilon_t)^{\mathcal{I}-1}]}{E[\mathcal{I}(1 - \epsilon_t)^{\mathcal{I}-1}]}. \end{aligned} \tag{42}$$

Applying (42), we compute $E[\mathcal{O}(X_t)]$ for all t iteratively. Note that (42) skips values for some t's, which are interpolated from the neighboring values. We then substitute $E[\mathcal{O}(X_t)]$ into (41) to compute $E[\phi(F_t)]$ for all t. Fig. 13(a) compares this model against simulations and confirms its accuracy. Our



(a) correlated degree



(b) uncorrelated degree

Fig. 14. Verification of model (44) in FRN with Zipf $\mathcal{I}$ and $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10$K.

second case involves uncorrelated in/out-degree, where we can estimate the size of the frontier by dividing its total out-degree by $E[\mathcal{O}]$:

$$E[\phi(F_t)] = \frac{E[\mathcal{O}(F_t)]}{E[\mathcal{O}]} = nE[1 - (1 - \epsilon_t)^{\mathcal{I}}] - \frac{t}{E[\mathcal{O}]}. \quad (43)$$

Note that we get the same result as (43) after expanding the recursion in (41) and using $\sum_{\tau=1}^{t} p(t) = E[\phi(S_t)]$. In Fig. 13(b), we compare model (43) against BFS on the uncorrelated-degree graph. Observe that Fig. 13(b) closely follows Fig. 12(b) since they are scaled versions of each other.

Our third crawl method, which we call *Frontier RaNdomization* (FRN), picked candidates randomly from the frontier. In contrast to BFS, FRN avoids back-to-back hits against the same website and achieves better politeness guarantees during large-scale web crawling [18]. Since any node in the frontier is equally likely to be picked, $E[\mathcal{O}(X_t)]$ in FRN is simply the average out-degree of the nodes in F_t. Hence,

$$E[\mathcal{O}(X_t)] = \frac{E[\mathcal{O}(F_t)]}{E[\phi(F_t)]}. \quad (44)$$

We use $E[\mathcal{O}(X_t)]$ from (44) in (41) to compute the $E[\phi(F_t)]$ model for all t iteratively. Then, we compare the result against FRN simulation in Fig. 14. Part (a) of the figure shows the correlated case, which is skewed to the left compared to Fig. 13(a) under BFS. Since both curves have the same $E[\mathcal{O}(F_t)]$, we can conclude that FRN still exhibits degree-bias for crawled nodes, but due to randomization of the frontier, the bias is less prominent compared to BFS. The uncorrelated case in Fig. 14(b) is, as expected, almost identical to the uncorrelated scenario in BFS.

Observe in Fig. 13(a) that BFS grows the frontier set to 70% of n. In contrast, FRN's frontier does not exceed 55% of n as shown in Fig. 14(a). This can be explained by the fact that BFS discovers unique nodes at a higher rate and grows its queues faster compared to FRN. As a result, FRN is not only more polite, but also more efficient in resource usage.

VI. Conclusion

We proposed an accurate analytical framework for characterizing applications that process random data streams, including such properties as the probability of uniqueness for discovered keys, the number of unique values accumulated by a certain time t, the average degree of seen nodes, and the size of the frontier during crawls on large-scale graphs under three different strategies. We demonstrated that these models were applicable not just to synthetically generated streams, such as those produced by BFS on random graphs, but also real workloads stemming from LRU caching and MapReduce processing of IRLbot and WebBase graphs.

References

[1] L. Becchetti, C. Castillo, D. Donato, A. Fazzone, and I. Rome, "A Comparison of Sampling Techniques for Web Graph Characterization," in *Proc. ACM LinkKDD*, Aug. 2006.

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

[3] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph Structure in the Web," *Computer Networks*, vol. 33, pp. 309–320, Jun. 2000.

[4] H. Che, Y. Tung, and Z. Wang, "Hierarchical Web caching systems: modeling, design and experimental results," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 7, pp. 1305–1314, Sep. 2002.

[5] A. Dan and D. Towsley, "An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 18, no. 1, pp. 143–152, May 1990.

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

[7] M. Gallo, B. Kauffmann, L. Muscariello, A. Simonian, and C. Tanguy, "Performance evaluation of the random replacement policy for networks of caches," in *Proc. ACM SIGMETRICS*, Jun. 2012, pp. 395–396.

[8] P. R. Jelenkovic, "Asymptotic Approximation of the Move-to-Front Search Cost Distribution and Least-Recently Used Caching Fault Probabilities," *The Annals of Applied Probability*, vol. 9, no. 2, pp. 430–464, May 1999.

[9] E. Krevat, T. Shiran, E. Anderson, J. Tucek, J. J. Wylie, and G. R. Ganger, "Applying Performance Models to Understand Data-Intensive Computing Efficiency," CMU, Tech. Rep. CMU-PDL-10-108, May 2010. [Online]. Available: http://www.pdl.cmu.edu/PDL-FTP/Storage/CMU-PDL-10-108.pdf.

[10] M. Kurant, A. Markopoulou, and P. Thiran, "Towards Unbiased BFS Sampling," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1799–1809, Oct. 2011.

[11] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, "IRLbot: Scaling to 6 Billion Pages and Beyond," in *Proc. WWW*, Apr. 2008, pp. 427–436.

[12] S. H. Lee, P.-J. Kim, and H. Jeong, "Statistical Properties of Sampled Networks," *Phys. Rev. E*, vol. 73, p. 016102, Jan. 2006.

[13] 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.

[14] L. Lovász, "Random Walks on Graphs: A Survey," in *Combinatorics, Paul Erdös is Eighty*, D. Miklós et al., Ed. János Bolyai Mathematical Society, Budapest, 1996, vol. 2, pp. 353–398.

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

[16] M. Molloy and B. Reed, "A Critical Point for Random Graphs with a Given Degree Sequence," *Random Structures and Algorithms*, vol. 6, no. 2/3, pp. 161–180, Mar./May 1995.

[17] M. Najork and J. L. Wiener, "Breadth-First Search Crawling Yields High-Quality Pages," in *Proc. WWW*, May 2001, pp. 114–118.

[18] V. Shkapenyuk and T. Suel, "Design and Implementation of a High-Performance Distributed Web Crawler," in *Proc. IEEE ICDE*, Mar. 2002, pp. 357–368.

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

[20] Yahoo Hadoop. [Online]. Available: http://developer.yahoo.com/hadoop/.

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