**CS 559: Algorithmic Aspects of Computer Networks** **Fall 2007**

# Lecture 13 — October 22, 2007

*Lecturer: John Byers* *BOSTON UNIVERSITY* *Scribe: Stephen Chao*

In today's lecture, we wrapped up our discussion of Flajolet-Martin sketches and briefly covered min-wise permutations. In the remaining time we began looking at random trees and consistent hashing.

## 13.1 Flajolet-Martin Sketches (reprise)

### 13.1.1 Algorithm

Given a set of items $S = \{i_1, i_2, \ldots, i_k\}$, a random, binary hash function $h$, and an $n$-bit array, we can construct an approximate representation of $S$ by the following method: For each $i$ in $S$, we repeatedly sample from $h$ until a 1 is returned, at which point we set the bit in the array whose index value is equal to the number of samples taken.

### 13.1.2 Example

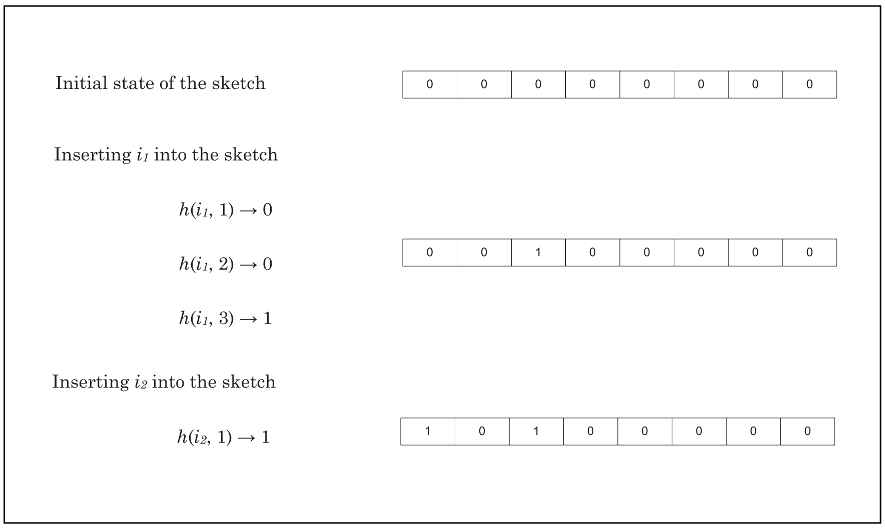
Initial state of the sketch
0 0 0 0 0 0 0 0
Inserting $i_1$ into the sketch
$h(i_1, 1) \rightarrow 0$
$h(i_1, 2) \rightarrow 0$
$h(i_1, 3) \rightarrow 1$
0 0 1 0 0 0 0 0
Inserting $i_2$ into the sketch
$h(i_2, 1) \rightarrow 1$
1 0 1 0 0 0 0 0

**Figure 13.1. An example illustrating the insertion of the first two elements in $S$.**

### 13.1.3 Observations

We note that size of the bit arrays are generally very small - the probability that $h$ returns more than $3 \cdot \log k$ 0s in a row is already quite small, so using $n = 4 \cdot \log k$ bits is considered safe.

Once we have constructed a sketch, there are two values we can use to estimate the number of elements in $S$:

- Estimator 1: The left-most zero in the array
- Estimator 2: the right-most one in the array

The simplest way to record the right-most one is to write down the index of its location, which takes $\log(4 \cdot \log k) = O(\log\log k)$.

## 13.2 Min-wise permutations [Broder et al. '00]

### 13.2.1 Introduction

Up to this point, we have seen methods for storing and retrieving approximate answers to various queries on sets, including the following:

- Membership queries (Bloom filters)
- Statistics of members (count-min summaries, multistage filters)
- Approximate set magnitudes (Flajolet-Martin)

Now, suppose we wanted to compare the similarity (or dissimilarity) between two sets. One situation where this need may arise is when a node in a P2P network is looking for dissimilar nodes to swap file chunks with. Another example appears in cache synchronization; when a cache is recovering after a crash, it will be interested in finding out how stale or fresh its contents are with respect to its neighbors.

So, what about using Bloom filters? The problem that arises is that deviations can grow quickly. Adding one element to the bloom filter can lead to the addition of 0 to $k$ new bits, where $k$ is the number of hash functions. Alternatively, we can use min-wise independent permutations as described in the next subsection.

### 13.2.2 Algorithm

Given a universe of keys $\mathbb{U}$, $A, B \subseteq \mathbb{U}$, and a permutation function $\Pi(\cdot)$, we compare the minimum elements of $\Pi(A)$ and $\Pi(B)$ such that if they agree, we conclude that the element we've selected is in $(A \bigcap B)$, otherwise the global minimum is in $(A \bigoplus B)$.

For the set of keys that we want to compare (say $A$) we take the minimum of $k$ independent permutations to produce a vector of minima totaling $k \cdot \log \mathbb{U}$ bits. These bits then are transmitted to $B$ or some intermediary that has the vector of minima for $B$, from which the "resemblance" of the two sets can be determined. The resemblance $r$ is defined in [2] to be

$$r(A,B) = \frac{S_A \bigcap S_B}{S_A \bigcup S_B}$$

where $S_A$ and $S_B$ are the vector of minima for $A$ and $B$ respectively.

## 13.3 Consistent Hashing and Random Trees [Karger et al. '97]

### 13.3.1 Introduction

Load balancing is an issue that arises in many types of networks, ranging distribution of data in P2P networks, to downloading some desired piece of data from a central server located on the web. We consider the following setup: there exists a central origin server containing a large set of items that users wish to request, and a network of caches that each contain some subset of the data contained at the origin server. Due to either either bandwidth or memory constraints, the caches can only duplicate a partial set of the origin server's data at any given point in time. The question is then, how should each cache choose the set of items to store so as distribute the requests across the caches and origin server as evenly as possible, given that users may not have access to all caches and will want to query the leasti-loaded nearby cache?

There are two main components to the solution given in [3]; we begin to describe these in the next two sections.

### 13.3.2 Random trees

The basic idea to begin with is that popular (more frequently requested) files should be distributed along the tree, while less popular ones should be kept closer to the origin server. To do this, each cache maintains a count of the number of times files (including those that the cache currently does not hold) have been requested by the end terminals (clients). After an unheld file has been requested more times than some threshold value, the file is pushed down the tree, thereby limiting the number of requests that can be propagated up the tree in the future due to that file.

The issues that arise here are:

- How do clients know what level to start querying at?
- How can the clients find out the locations of leafs and inner nodes in the cache network?

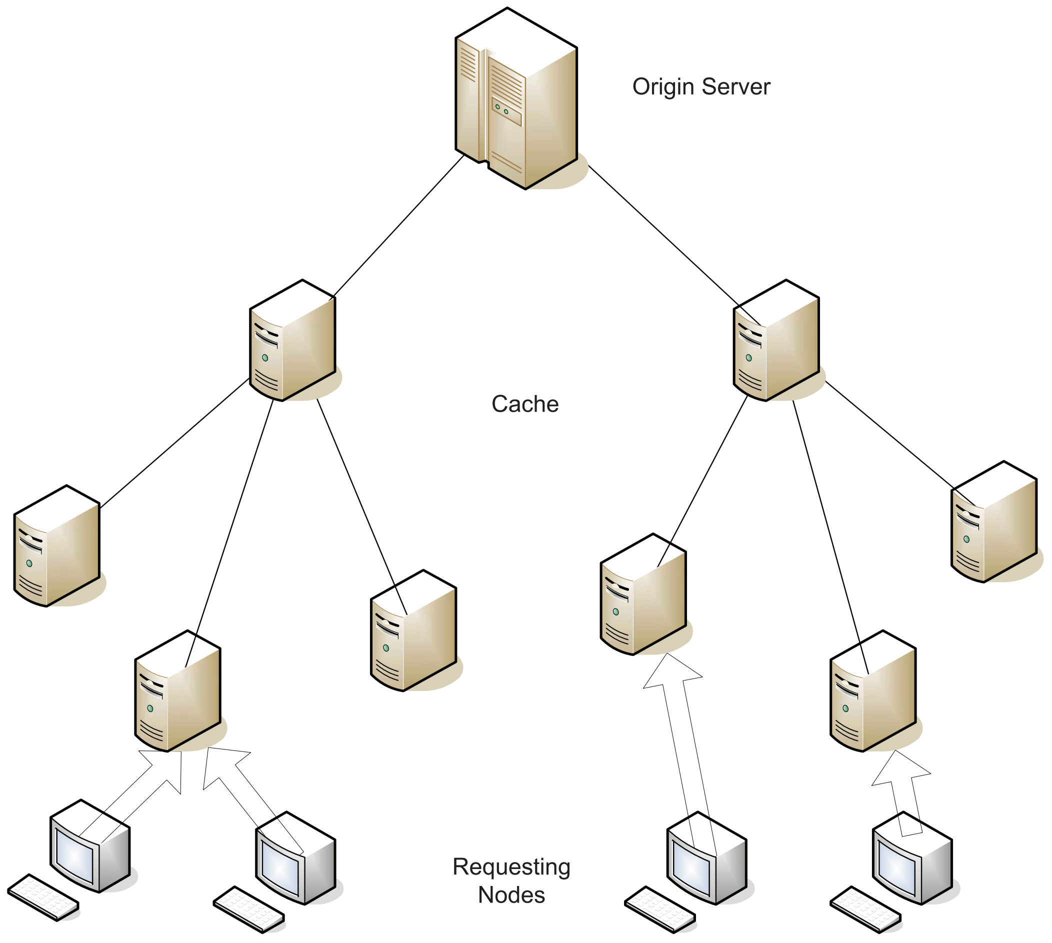

**Figure 13.2. A random tree consisting of an origin server and 8 distributed caches.**

## 13.3.3 Consistent hashing

First, some definitions:

- **items: things to locate**
- **buckets: location of items**
- **views: arbitrary subset of buckets**

Our first goal is to provide guarantees on load, i.e. have items hash uniformly across buckets. This way every bucket contains roughly the same number of items , which would in the ideal case be minimal. Furthermore, we also want to associate one bucket with a given item for each view (where there may be more than one bucket with the same item).

In the next lecture, cover the specifics of how this can be realized through hashing of buckets and items onto the real number line.

# Bibliography

[1] P. Flajolet and G. N. Martin. "Probabilistic counting algorithms for data base applications," in *Journal of Computer and System Sciences*, 31(2):182-209, October 1985.

[2] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. "Min-wise independent permutations," in *Journal of Computer System Sciences 60*, 3(2000), pp. 630-659.

[3] D. Karger, E. Lehman, F. T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in *Proceedings of the* $29^{th}$ *ACM Symposium on Theory of Computing* (El Paso, TX, May 1997), pp. 654-663.