Real Time Ranking of Popular Items

Jeremy Calvert
CSEP 521: Applied Algorithms
March 12, 2007

Motivation

The community based news site digg.com provides most 'dugg' news stories in the last hour, week, and month per predefined category. A product I've been working on has a requirement for similar functionality. A couple of colleagues of mine and I came up with a way to do this that we thought would scale. We didn't precisely define storage limitations or query efficiencies, but we knew we wanted to handle up to 1000s of data points per second, and provide near real-time responses to queries on the "top 100 items in the last hour, week, or month". This paper is an exploration of existing published work to this end.

Digg

I couldn't find precise descriptions of digg's algorithm, but I found a couple of discussions about it. They use rules to prevent users from "gaming" the system and describe no other details of their algorithm on the basis that doing so help people game the system. This simple and novel interpretation of user data has sparked considerable interest, and even some controversy. I'm interested in seeing if a scalable ranking algorithm need be complicated, and if there are measures other than secrecy that one could use to prevent gaming.

Considerations

In the vocabulary of literature on the topic, this is a problem of *counting frequencies* of items in a *data stream*. We can't do these rankings with completely accurately for arbitrary input rates. So first, we have to formalize the uncertainties involved in an approximation. The algorithm should be able to continuously process the data (as it arrives in large volumes, amortizing update costs), store summary data in memory (no disk accesses for queries), and execute the queries with little complexity (complexity can't grow with *N*, the net size of the input).

Stanford

A lot of interesting, relevant work in the area has come out of the Stanford Stream Data Manager project (http://infolab.stanford.edu/stream/). The paper[3] I'll focus on was written by two Ph.D. students, one now at Google, the other at Microsoft Research. The site includes a paper [1] describing an extension of SQL called Continuous Query Language, or CQL, that Oracle reportedly will soon be selling. The site also offers source code for running a DSMS (Data Stream Management System) server and a CQL client. This suggests that the subject matter is getting increased attention in a variety of contexts.

Definitions

Let N denote the number of elements in a data stream. An ϵ-approximate frequency count is a set of pairs $(e, \bar{f}_e)$. Here e is an element that occurs in the data stream (e.g., a URL), and $\bar{f}_e$ is an approximation of the number of occurrences of e in N (e.g., number of times that URL was clicked). They satisfy the properties

1. If e's true frequency f_e exceeds ϵN, then it is in the output. (i.e. no false negatives in top results queries)
2. $f_e - \epsilon N \le \bar{f}_e \le f_e$ (the approximation is less than the true value, but not by too much)

We're interested in a so-called *frequent elements* problem which is, given a threshold parameter s, find the set of elements whose frequency exceeds sN. In this problem, the first property becomes: If e's true frequency f_e exceeds $(s - \epsilon)N$, then it is in the output. Here we assume $s \ge \epsilon$.

Related query: Quantiles
A interesting related problem is finding *quantiles*. Finding the k^{th} largest of N elements requires at least $1.5N$ comparisons and at most $5.43N$[5]. So, for example, finding the exact median element in a single pass requires storing $N/2$ elements. For data streams, neither this complexity or storage requirement is acceptable. So the notion of an ϵ-estimate quantile was defined in [6] . An element e is said to be an ϵ-approximate φ-quantile if its rank is between $\lceil(\varphi-\epsilon)N\rceil$ and $\lceil(\varphi+\epsilon)N\rceil$, (here $\varphi\in[0,1]$) . While [3] derives parallel results for both frequency counting and quantile finding, from here on we'll focus on the former.

Sliding window
We're distinctly interested in counts over a fixed sample, wherein the notion of a sliding window is instrumental. The authors of [3] stated they "believe that no previous work on space-efficient approximate counts over sliding windows exists." A sliding window is a subset of N elements, such that at each step a user can insert an element at the beginning (as with any data stream) or delete an element at the end. A window can be of variable or fixed size. Fixed means that once it reaches a predefined size, each insertion is followed by one deletion.

Time-based sliding
For our problem, we want to base the size of the sliding window on the clock time at which an element was inserted into the stream. So we opt for a variable sized sliding window. When an element reaches a certain age, it's deleted. It's easy to confuse the step corresponding to the insertion of an element into the stream with clock time. It's helpful to think of the former as a counter cycle, one per click, and to keep in mind that it's not tied in any way to clock time.

Bounded vs. Unbounded window
If we allow the window to be variable sized, do we allow it to grow without a predetermined bound? It turns out that techniques are simpler with bounded windows, and that these can be adapted to unbounded windows. For us, we know that we want to bound the window in time (to say, a week), so if we we bound by $W=2^{32}$, then we can handle an average click rate of ~7K clicks/second. This seems reasonable, at least for a first go.

Negative input
Another variation of data stream approximations allow for deletions in the front of the data stream. For us, this could model a user unclicking or "burying" an item. Unfortunately, our methods do not adapt at all to this scenario. Work in [4] addresses negative input using entirely different ideas from group testing and error-correcting codes. However, [4] doesn't consider such approximations over sliding windows.

Aggregate queries
If we wanted, for example, to count and rank sites based on clicks made on any page hosted by that site, we would use so-called *iceberg queries*, or some equivalent, that compute aggregate functions (e.g. sums) of counts, over sets of elements sharing an attribute (site in our example). Most of the approximation techniques discussed can be generalized to support these.

Deterministic vs. Probabilistic
The approximation algorithms presented in [3] come in deterministic and probabilistic varieties. The probabilistic approaches have the attractive distinction that space requirements are not tied to the number of data points in stream (or the bound on the size of the window). Let δ denote the rate of failure, a failure being that one of the two defining properties of an ϵ-approximate frequency count does not hold, then "in a random sample of size $O(\frac{1}{\epsilon^2}\log(\epsilon\delta)^{-1})$ the relative frequency of any element in the sample differs from its true frequency in the base dataset by at most ϵ."[3] This leads to

a sampling scheme that requires $O(\frac{1}{\epsilon}\log(\epsilon\delta)^{-1})$ space. But these methods are fairly complex, and perhaps overkill for our particular situation. I'm going to omit further discussion of them.

Our Algorithm

I implement the deterministic algorithm that estimates frequency counts over a sliding, variable sized, bounded window. This is covered in section 5 of [3]. The key notion is of a Sketch, which is essentially an ϵ-approximate frequency count over a subset of the stream. It has an insert method, like any stream, and keeps approximate counts of the most common elements in that subset.

We'll assemble multiple layers of such sketches as in the diagram shown from [3], sketches on the same level, once fully populated, are of equal size. They're also disjoint and exhaustive (i.e., tiled). Remember, here size of a sketch is number of underlying data points, not clock-time duration. Completed sketches on level i are double the size of those on level $i-1$. All dimensions are a function of the value of ϵ chosen for the approximation. Here, the value N indicates the number of elements in our sliding window, and W is the upper bound on N.

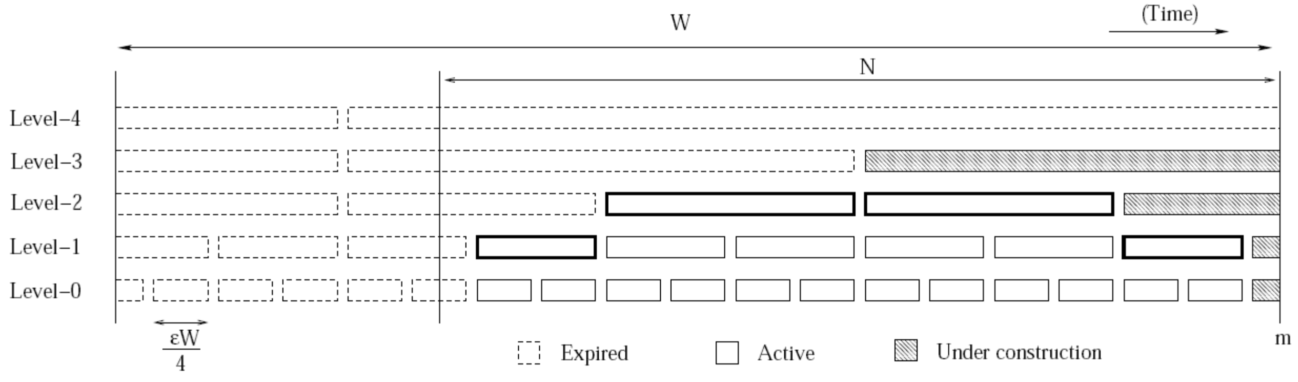

Figure 1: Levels and Blocks used in $\mathcal{C}_{W,\epsilon}(m,N)$ and $\mathcal{Q}_{W,\epsilon}(m,N)$

The summary data associated to each Sketch consists of the structure used in the GriesMisra [2] algorithm. GriesMisra is essentially the observation that you can find all elements that occur at least $N/(k+1)$ times using at map with at most k entries. To run it, you:

Maintain k items, and their counters.

If next item x is one of the k, increment its counter.

Else if there is an entry with a zero counter, put x there with count = 1

Else (all counters non-zero) decrement all k counters

The choice of 4 in declaring the level-0 blocks to be of size $\frac{\epsilon W}{4}$ was at the author's discretion. One could choose other values if the following are recomputed.

$L=\log_2(4/\epsilon)$ The highest level

$\epsilon_l=\frac{\epsilon}{2(2L+2)}2^{L-l}$ The error at level l

$N_l=\frac{\epsilon W}{4}2^l$ Size of level l block

To run the algorithm, you maintain the sketches on every level. When the nth sample is added, the latest Sketch on level l graduates from “Under Construction” to “Active” (and a new Under Construction Sketch of size 0 is added) iff $n \bmod N_l=0$.

The storage required for a MisraGries structure at level l is $k = 1/\epsilon_l$, and at this point, we have all the information we need to compute the total space requirements for this estimation. We're going to omit the computation here, but Theorem 1 in section 5.1 of [3], shows that it uses $O(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon})$ space.

A final detail is in maintaining our sliding window on the basis of physical time. To do this, we maintain a time stamp for each Sketch on level-0. It *should* be largely irrelevant which entry corresponds to this time stamp. If it's not, then it's taking too much physical time to obtain the $\frac{\epsilon W}{4}$ samples, meaning our choice of W is too large to be effective on our stream. It makes sense that in order to use a bounded window approximation for a time-based window, we need some understanding of the data stream's average volume in time.

For this initial implementation, we'll take the time stamp to be the time that the first sample was added to the sketch. So at each step, we check the time stamp of the oldest active level-0 sketch, and if it's longer ago than our time-based sliding window width, we expire it, and any higher level, non-expired sketches directly above it.

With that, we have all the pieces we need to put together an efficient topN estimator and test it out. The appendix shows a 250LOC implementation of the algorithm as described above.

To Do

First, I need to verify that it does in fact provide ϵ-approximate frequency counts, and that this particular implementation doesn't have additional space requirements.

It's interesting that there's no apparent reason we would need to repeat any of the Sketch data for different time periods. So, we could generalize the definition of a Sketch to have an array of states, one per time period over which we might want to query. Then, when we query, we specify a time interval, and where we initialize `activeIdxs` in `ArasuManku.getTopActiveSketches()`, we set it to the oldest Sketch on that layer that is not expired for the provided time interval. Having these queries available for a family of time intervals could allow for some interesting user interfaces.

Bibliography

[1] A. Arasu and S. Babu and J. Widom The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal.* 15 (2), pages 121 – 142, June 2006

[2] J. Misra and D. Gries. Finding repeated elements. *Sci. Comput. Programming*, 2(2), pages143-152, Nov. 1982

[3] A. Arasu and G. Manku, Approximate Counts and Quantiles over Sliding Windows, In *Proc. of the 23rd ACM SIGACT-SIGMOD-SIGART Symp. on the Principles of Database Systems,* pages 282-296, June 2004

[4] G. Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. In *Proc. of the 22nd ACM SIGACT-SIGMOD-SIGART Symp. on the Principles of Database Systems,* pages 296-306, June 2003

[5] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448-461, Aug. 1973

[6] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one

pass and with limited memory. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 426-435, June 1998.

Appendix:

```
/**
 * Deterministic time-based, sliding window approximation of the top occurring
 * items in a data stream.
 *
 * Based on Section 5 of "Approximate Counts and Quantiles over Sliding Windows"
 *
 * @param <Value>
 *           The class of values being ranked
 */
public class ArasuManku<Value> {

        private int L = 16;    // The number of levels we use. We cast 2^L to an
                               // integer, so need to respect that bound.
        private int M = 32;    // The log of the capacity of the window we allow
        private long n = 0;    // The data series ticker
        private long oneOverEpsilon = (long) Math.pow(2, L - 2);
        private long W = (long) Math.pow(2, M);
        private long period = 7 * 24 * 60 * 60 * 1000;

        List<List<Sketch>> activeSketches = new ArrayList<List<Sketch>>();
        List<Sketch> underConstruction = new ArrayList<Sketch>();

        /**
         * @param L   The number of levels we use (minus 1).
         * @param M   The log of the total number of samples that we can handle
         *            within the period.
         * @param period   The number of milliseconds in the period over which we're
         *            tracking the topN.
         * @param time The initial time that this stream is being monitored. In
         *            non-test scenario, should be System.currentTimeMillis() in
         *            first run, and persisted between consecutive runs
         */
        public ArasuManku(int L, int M, long period, long time) {
                this.L = L;
                this.M = M;
                for (int l = 0; l <= L; l++) { // Initialize levels
                        underConstruction.add(l, new Sketch(l, n, time));
                        activeSketches.add(new ArrayList<Sketch>());
                }
        }

        /**
         * Insert a value in the data stream at time t
         *
         * @param v The value of the data stream element
         * @param time The physical time that this entrie is being added. Subsequent
         *            calls must have non-decreasing values of time. In non-test
         *            scenarios, should be System.currentTimeMillis()
         */
        public void insert(Value v, long time) {
                n++;
                for (int l = 0; l <= L; l++) {
                        Sketch s = underConstruction.get(l);
                        if (s.insert(v, n)) { // If we graduated this sketch
                                underConstruction.remove(l);
                                underConstruction.add(l, new Sketch(l, n, time));
                                activeSketches.get(l).add(s);
                        }
                }
                expireOld(time);
        }

        private void expireOld(long time) {
                List<Sketch> level0 = activeSketches.get(0);
                if (level0.size() < 1) {
                        return;
                }
                Sketch oldestLevel0 = level0.get(level0.size() - 1);
                if (time - oldestLevel0.creationTime > period) {
```

```
			long expiredIndex = oldestLevel0.streamIndex;
			for (int l = 0; l < activeSketches.size(); l++) {
				List<Sketch> level_l = activeSketches.get(l);
				if (level_l.size() > 0
					&& level_l.get(level_l.size() - 1).streamIndex <= expiredIndex) {
					level_l.remove(level_l.size() - 1);
				}
			}
		}
	}

	/**
	 * Returns a map of elements to approximate frequency counts, for all elements occuring over 1/N
	 * @param time
	 */
	public Map<Value, Integer> getTopN(long time) {
		expireOld(time);
		Map<Value, Integer> topN = new HashMap<Value, Integer>();
		for (Sketch sketch : getTopActiveSketches()) {
			for (Entry<Value, Integer> entry : sketch.mg.counts.entrySet()) {
				int newCount = entry.getValue();
				if (topN.containsKey(entry.getKey())) {
					newCount += topN.get(entry.getKey());
				}
				topN.put(entry.getKey(), newCount);
			}
		}
		return topN;
	}

	private List<Sketch> getTopActiveSketches() {
		List<Sketch> topSketches = new ArrayList<Sketch>();
		List<Integer> activeIdxs = new ArrayList<Integer>();
		for (int i = 0; i <= L; i++) {
			activeIdxs.add(i, activeSketches.get(i).size() - 1);
		}
		boolean complete = false;
		int l = 0;
		while (!complete) {
			if (activeIdxs.get(l) >= 0) {
				if (l < L
					&& activeIdxs.get(l + 1) >= 0
					&& activeSketches.get(l).get(activeIdxs.get(l)).streamIndex
					<= activeSketches.get(l + 1)
							.get(activeIdxs.get(l + 1)).streamIndex) {
					l++;
				} else {
					topSketches.add(activeSketches.get(l)
							.get(activeIdxs.get(l)));
					stepActiveIdxs(l, activeIdxs);
				}
			} else { l--; }
			if (l == 0 && activeIdxs.get(l) < 0) {
				complete = true;
			}
		}
		return topSketches;
	}

	private void stepActiveIdxs(int l, List<Integer> activeIdxs) {
		for (int i = l; i >= 0; i--) { // Step
			activeIdxs.set(i, (int) (activeIdxs.get(i) - Math.pow(2, l - i)));
		}
	}

	class LevelBlock {
		int l; // level
		long oneOverEpsilon_l; // 1 over error at this level
		long N_l; // size of block

		LevelBlock(int l) {
			oneOverEpsilon_l = (long) (2 * (2 + L) * Math.pow(2, l + 2)); // need
			N_l = (long) Math.pow(2, M - L - 2 - 2 + l);
		}

	}
```

```
        enum SketchState { UNDER_CONSTRUCTION, ACTIVE, EXPIRED; }

        class Sketch {
                SketchState state;
                long streamIndex; // The first index in the block
                LevelBlock levelBlock; // The size of the block
                MisraGries<Value> mg; // The top items
                long creationTime; // Used for time-based deletion, only used by level-0 Sketches

                Sketch(int l, long streamIndex, long creationTime) {
                        state = SketchState.UNDER_CONSTRUCTION;
                        this.streamIndex = streamIndex;
                        this.levelBlock = new LevelBlock(l);
                        this.mg = new MisraGries<Value>(levelBlock.oneOverEpsilon_l);
                        this.creationTime = creationTime;
                }

                // Inserts into underlying MisaGries, returns true if this Sketch Activates
                boolean insert(Value v, long n) {
                        mg.insert(v);
                        if (levelBlock.N_l <= n - streamIndex) {
                                state = SketchState.ACTIVE;
                                return true;
                        } else {
                                return false;
                        }
                }
        }
}

public class MisraGries<Value> {

        Map<Value, Integer> counts = new HashMap<Value, Integer>();
        long k; // == 1/epsilon

        public MisraGries(long k){
                System.out.println(k);
                this.k = k;
        }

        public void insert(Value x){
                if(counts.containsKey(x)){
                        counts.put(x, counts.get(x) + 1);
                } else if (counts.size() < k) {
                        counts.put(x, 1);
                } else {
                        Iterator<Value> it = counts.keySet().iterator();
                        while(it.hasNext()){
                                Value v = it.next();
                                if(counts.get(v) == 1){
                                        it.remove();
                                } else {
                                        counts.put(v, counts.get(v) - 1);
                                }
                        }
                }
        }
}
```