Uncertainty in Aggregate Estimates from Sampled Distributed Traces

Nate Coehlo, Arif Merchant, Murray Stokely
{natec,aamerchant,mstokely}@google.com
Google, Inc.

Abstract

Tracing mechanisms in distributed systems give important insight into system properties and are usually sampled to control overhead. At Google, Dapper [8] is the always-on system for distributed tracing and performance analysis, and it samples fractions of all RPC traffic. Due to difficult implementation, excessive data volume, or a lack of perfect foresight, there are times when system quantities of interest have not been measured directly, and Dapper samples can be aggregated to estimate those quantities in the short or long term. Here we find unbiased variance estimates of linear statistics over RPCs, taking into account all layers of sampling that occur in Dapper, and allowing us to quantify the sampling uncertainty in the aggregate estimates. We apply this methodology to the problem of assigning jobs and data to Google datacenters, using estimates of the resulting cross-datacenter traffic as an optimization criterion, and also to the detection of change points in access patterns to certain data partitions.

1 Introduction

Estimates of aggregated system metrics are needed in large distributed computing environments for many uses: predicting the effects of configuration changes, capacity planning, performance debugging, change point detection, and simulations of proposed policy changes. For example, in a multi-datacenter environment, it may be desirable to periodically "repack" users and application data into different datacenters. To evaluate the effects of different rearrangements of data, it is necessary to estimate several aggregates, such as the cross-datacenter traffic created and the aggregate CPU and networking demands of the applications placed in each datacenter. Most organizations deploy telemetry systems [4, 5] to record system metrics of interest but, quite often, we find that some of the metrics required for a particular evaluation were not recorded. Sometimes this occurs because the number of possible metrics is too large for the problem of interest, other times it can be due to difficult implementation and a lack of perfect foresight.

At Google, we deploy additionally a ubiquitous tracing infrastructure called *Dapper* [8] that is capable of following distributed control paths. Most of Google's interprocess communication is based on RPCs and Dapper samples a small fraction of those RPCs to limit overhead. These traces often include detailed annotations created by the application developers. While the primary purpose of this infrastructure is to debug problems in the distributed system, it can also be used for other purposes like monitoring the network usage of services and the resource consumption of users in our shared storage systems.

From any system with sampling, such as Dapper, Fay [7], or the Dapper-inspired Zipkin [1], it is straightforward to get an estimate for aggregate quantities. However, assessing the uncertainty of an estimate is more involved, and the contribution of this work is finding covariance estimates for aggregate metrics from Dapper, based on the properties of Dapper sampling mechanisms. Given these covariance estimates, one can attach confidence intervals to the aggregate metrics and perform the associated hypothesis tests.

In Section 2 we summarize the relevant statistical properties of sampling in Dapper. Section 3 gives the estimation framework and algorithm for covariance estimation, while the proof appears in Appendix A. Section 4 gives two case studies of statistical analysis about distributed systems using this framework, and Section 5 has concluding remarks.

2 Background on RPC Sampling

Estimating an aggregate quantity from a sampled system is simple; for each measured RPC you have a result and a sampling probability, so summing those results weighted by the inverse of their sampling probability will give an unbiased estimate of the quantity of interest. To calculate the uncertainty (variance) of such an estimate, however, it requires knowledge of the joint sampling probability of any two RPCs, so a more detailed understanding of the sampling mechanism is necessary.

RPCs in distributed systems can be grouped in terms of an initiator, and we refer to this grouping as a *trace*. Each trace at Google is given an identifier (**ID**), which is an unsigned 64-bit integer, and all RPCs within a trace share that one ID. The ID is selected randomly over the possible values so collecting RPCs when $\mathbf{ID} < (2^{64}-1)*s$ will induce a sampling probability of s, which we call the **Server Sampling Probability**. In addition, as explained in Section 4.6 of [8], an independent sampling stage can occur at some nodes which reduces the RPCs collected, and we refer to this here as **Downsampling**. Downsampling is based on a hash of the trace ID, and makes the further requirement that $\mathbf{hash(ID)} < (2^{64}-1)*d$, for downsampling factor d. In effect, each trace ID can be mapped to a point (s',d') on the the unit square, the distribution of those mapped points is uniform, and an RPC within a trace is included if that node has $s' \leq s$ and $d' \leq d$. Figure 1 shows an example trace and lists the possible RPCs returned based on the value (s',d') drawn. Traces with several sampling properties often arise when the execution path spans many layers of infrastructure, since different levels may have been configured differently by developers, and since downsampling based on system pressure may be present in some places and not others.

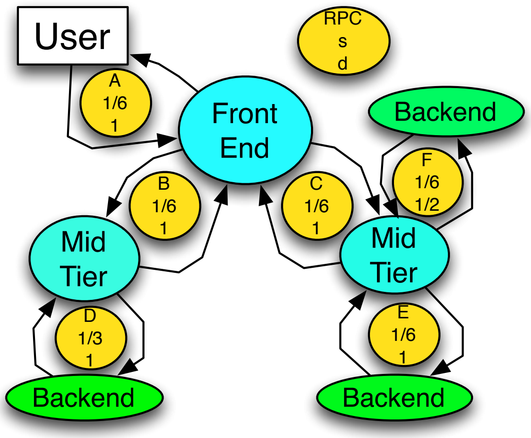

Figure 1: Trace representation, where different subsets of RPCs A-F will be returned depending on the value of trace ID $\rightarrow (s',d')$. If $s' > 1/3$ then none are returned, and if $1/6 < s' \leq 1/3$ then only D is returned. If $s' \leq 1/6$ and $d' \leq 1/2$ then all RPCs are returned. If $s' \leq 1/6$ and $d' > 1/2$ then all except F are returned.

3 Estimation Results and Algorithms

Suppose we want to estimate a system quantity of interest, θ, which can be represented as a sum of a function of the RPCs in a distributed system. Given a sample of RPCs available as described in the previous section, we find $\hat{\theta}$, an unbiased estimate of θ, and $\hat{\Sigma}$, and unbiased estimate of the covariance matrix of $\hat{\theta}$, where the later can be used to describe the uncertainty in our estimates of θ. The unbiasedness of $\hat{\theta}$ and $\hat{\Sigma}$ do not require any assumptions on the distribution of RPCs or on the server and downsampling factors. In detail:

- We represent RPC i by its trace ID, server and downsampling factors, and let λ represent all other information: $RPC_i = (ID_i, s_i, d_i, \lambda_i)$.
- We apply a function $f : \lambda \longrightarrow \mathbf{x}$ to get $(ID_i, s_i, d_i, \mathbf{x}_i)$
- Letting Ω represent all RPCs during our time period of interest, we have $\theta = \sum_{i \in \Omega} x_i$.
- Letting S be the sample returned by Dapper, and $\mathbf{1}_i$ be the boolean random variable representing whether RPC_i was included in the sample S, we get an unbiased estimate of θ from

$$\hat{\theta} = \sum_{i \in S} \frac{\mathbf{x}_i}{s_i * d_i} = \sum_{i \in \Omega} \frac{\mathbf{x}_i \mathbf{1}_i}{s_i * d_i}$$

- The algorithm GetSigmaHat below produces $\hat{\Sigma}$, an unbiased estimate of $\Sigma = \mathrm{Cov}\,\hat{\theta}$.

In this paper we will use the normal approximation for inference, $\hat{\theta} \sim \mathcal{N}(\theta, \hat{\Sigma})$, which is a generalization of normal approximation to the binomial distribution. However, we believe that the considering the variance adds substantial value and protection against false positives to any analysis involving these estimates, even in the case of small samples sizes with highly variable x_i, s_i, d_i where the normal approximation is not ideal.

The remainder of this section provides an outline for proving $\hat{\Sigma}$ is unbiased, gives a simple algorithm, and finds its complexity. The next section applies these results to the statistical analysis of real distributed systems.

3.1 Calculation Overview

To find the an unbiased estimate of the population covariance matrix Σ, we first find Σ, then appropriately weight sample quantities and show the result is unbiased. A detailed calculation is in the Appendix, and there it is divided by these three ideas:

1. Within a trace, the boolean random variables $\mathbf{1}_i$ and $\mathbf{1}_j$ must be the same if $s_i = s_j$ and $d_i = d_j$, so we can aggregate the values of x corresponding to the same (s,d) tuple to y in our representation of $\hat{\theta}$ and re-parameterize the problem in terms of the distinct values of (ID, s, d) and the sums y. This simplifies proof notation and improves the algorithm performance, as discussed in section 3.2.

2. Letting $y[j]_i$ denote component j of y for the (ID, s, d) tuple i, we have

$$\Sigma_{(j,k)} = \text{Cov}(\hat{\theta}_j, \hat{\theta}_k) = \sum_{i\in\Omega}\sum_{i'\in\Omega} \frac{x[j]_i x[k]_{i'}}{s_i s_{i'} d_i d_{i'}} \text{Cov}(\mathbf{1}_i, \mathbf{1}_{i'})$$

   so the problem reduces to finding the covariance between sampling any two tuples (ID_i, s_i, d_i) and $(ID_{i'}, s_{i'}, d_{i'})$.

   As described in Section 2, the trace ID is mapped to two independent uniform random variables on $(0,1)$, which we denote by (U_i, V_i) and assume they are independent across i. Therefore, if $ID_i \neq ID_{i'}$ then they are independent and the covariance is zero. If $ID_i = ID_{i'}$ then we must have [1]

$$\begin{aligned} \text{Cov}(\mathbf{1}_i, \mathbf{1}_{i'}) &= \mathbb{E}(\mathbf{1}_i \mathbf{1}_{i'}) - \mathbb{E}(\mathbf{1}_i)\mathbb{E}(\mathbf{1}_{i'}) \\ &= (s_i \wedge s_{i'}) * (d_i \wedge d_{i'}) - s_i s_{i'} d_i d_{i'} \end{aligned}$$

3. Finally, since the resulting population covariance matrix depends on cross terms within the same trace, weighting sampled cross-terms by their probability of inclusion, $(s_i \wedge s_{i'}) * (d_i \wedge d_{i'})$, will give an unbiased estimate.

3.2 Covariance Estimation Algorithm and Complexity

Algorithm GetSigmaHat returns $\hat{\Sigma}$, which is the sum of the contributions over each trace ID [2]:

Algorithm 1 GetSigmaHat

```
$M \leftarrow$ a $J \times J$ matrix of zeros.
for all $ID \in S$ do
    $M+ = ProcessSingleTrace(ID)$
end for
return M
```

While there may be a large number of RPCs within a trace, the number of distinct (s,d) tuples within a trace is small; across all traces we collected there are less than 20 distinct combinations. Given N_t RPCs within a trace and M_t distinct combinations, aggregating in the first step of ProcessSingleTrace before running the loop scales as $N_t log(M_t) + M_t^2 * J^2$ rather than $N_t^2 * J^2$. Letting $M = \max M_t$ and T be the number of traces, we sum over traces for the bound

$$\begin{aligned} \sum_t N_t log(M_t) + M_t^2 * J^2 &\leq (\sum_t N_t) log(M) + T * M^2 * J^2 \\ &= N log(M) + T M^2 * J^2 \end{aligned}$$

Since M is bounded by a small number in practice, we have linear scaling in the number of RPCs and Traces. In addition, one could split GetSigmaHat over several machines, sharding by trace ID, with each returning their component of the $J \times J$ covariance estimate.

Algorithm 2 ProcessSingleTrace

```
Given a collection of $(s_i, d_i, x_i)$ corresponding to a given ID, aggregate data over the unique tuples of $(s,d)$ to get $(s_k, d_k, \mathbf{y}_k)$ where $\mathbf{y}_k = \sum_{\{j|(s_j,d_j)=(s_k,d_k)\}} \mathbf{x}_j$ and we let $K_t$ be the number of distinct tuples resulting form this aggregation.

$M \leftarrow$ a $J \times J$ matrix of zeros.

for all $k \in 1 : K_t$ do
    for all $k' \in 1 : K_t$ do
        $w = \frac{1 - \max(s_k, s_{k'}) * \max(d_k, d_{k'})}{s_k s_{k'} d_k d_{k'}}$
        $M += w * (y_k \otimes y_{k'})$
    end for
end for
return M
```

4 Case Studies

4.1 Bin Packing and Cross-Datacenter Reads

Large scale, distributed computing environments may comprise tens of data centers, tens of thousands of users, and thousands of applications. The configuration of storage in such environments changes rapidly, as hardware becomes obsolete, user requirements change, and applications grow, placing new demands on the hardware. When new storage capacity is added — for example, by adding or replacing disks in existing data centers, or by adding new data centers — we must decide how to rearrange the application services, data, and users to take best advantage of the new hardware.

[1]We use the notation $min(a,b) = a \wedge b$ and $max(a,b) = a \vee b$.

[2]We denote the outer product between two vectors as $x \otimes y$.

An optimizer who bin-packs the application data and the users into the data centers will use data from many sources, may forecast growth rates of some parameters, and will try to satisfy various constraints. One component of such an optimization is to control the number of cross-datacenter reads that result from the packing, and simulation of this would require a full record of all user/application pairs. However, maintaining a complete record of the traffic for each user/application pair is prohibitively expensive, since there are millions of such pairs, so we can instead use the Dapper sampled traces of RPCs to estimate the cross-datacenter traffic for each scenario.

To illustrate the usefulness of our procedure for comparing policies over historical samples, we consider bin-packing user data in three nearby data centers. There is considerable work on the subject of file and storage allocation in the literature [6, 2, 3]. We do not claim to present optimal or useful bin-packing strategies here, but we do claim to be able to evaluate the comparative advantage in terms of cross-datacenter reads.

Data in a storage system is written by some user, which we call the *owner*, and is later read by that user, or potentially by many other users. We decide to pack data so each owner is only in one data center according to two strategies.

Strategy 1: basic
From a snapshot of the three cells, we figure out the total storage, and the percentage that goes to each user by adding up their contributions over the three cells. Then we partition the owners by alphabetical order so each datacenter gets $\frac{1}{3}$ of the total data.

Strategy 2: crossterm
This simple policy tries to put most cross user traffic in the first cell. We define the *adjacency* between two users as the estimated number of cross user reads divided by their combined storage capacity. We then allocate pairs of owners with the highest adjacency to the first cell until it reaches $\frac{1}{3}$ of the total data, then move on to the next cell.

4.1.1 Results

We compare the cross datacenter traffic for the two strategies above applied in simulation to three datacenters that each store several tens of petabytes of data belonging to over 1000 users. We use data collected from May 6, 2012 through May 12 to train the policy *crossterm*, then we evaluate the performance from period from March 25th through June 5th by assuming that a read by user A is initiated in the datacenter that stores data for user A. Our collection period from Dapper has millions of RPCs with a range of sampling probabilities extending down to $5e^{-7}$.

To test whether there is a significant difference between the resulting cross-datacenter reads, we look at the difference between the two estimates and form 95% confidence intervals around that difference, noting that when the interval does not cross zero it corresponds to rejecting the Null hypothesis that there is no difference between the strategies [3].

For each day, and for each policy, we get an estimate of the resulting cross datacenter traffic. To decrease our vulnerability to setting policy based on sampling aberrations, we test against the Null hypothesis that they both produce the same number of cross datacenter reads. In Figure 2 we show these intervals, where the y-axis has been scaled by the average for *basic* over the entire testing period; *crossterm* is significantly better than *basic* on every day, and we estimate it does over 20% better.

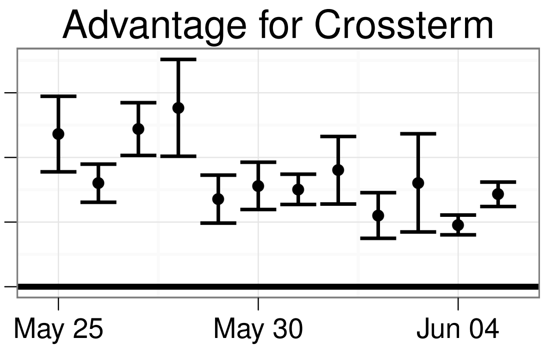

Figure 2: Estimate difference in number of daily cross datacenter reads, plus or minus two Standard Errors. The Y-Axis is normalized by the average number of cross datacenter reads for strategy *basic*.

4.2 Change Point Detection

It is often useful for a monitoring tool to detect sudden changes in system behavior, or a spike in resource usage, so that corrective action can be taken — whether by adding resources or by tracking down what caused the sudden change. In this case, we wanted to monitor the number of disk seeks to data belonging to a certain service, and to detect if the number of cache misses increased suddenly due to a workload change. The system logging available did not break out miss rates per service at the granularity we needed, but we could estimate the

[3]Here, the function $f : \lambda \to x$ a 2-vector of booleans indicating whether that RPC would have caused each strategy to result in a cross-datacenter read. The advantage for the *crossterm* strategy is estimated as $\hat{\theta}_1 - \hat{\theta}_2$, and the corresponding variance estimate is $\hat{\Sigma}_{1,1} + \hat{\Sigma}_{2,2} - 2\hat{\Sigma}_{1,2}$.

miss rates based on Dapper traces. However, we only want to detect *real* changes, and hoped to have few false positives induced by sampling uncertainty.

One alternative to alerting based on relative differences it to alert only if the difference is significantly different from zero. The problem with this approach is that some services have higher sampling rates, and given a high sampling rate, small true differences will be flagged as significant. Since we expect that there to be some true variation from day to day, we instead flag if we reject the null that the number of seeks increased by less than 10%.

In particular, let μ_t be the number of seeks on day t, and $\hat{\mu}_t$ be our estimated number [4] of seeks for day t, and

$$\begin{aligned} H_0 &: \frac{\mu_t}{\mu_{t-1}} \leq 1.1 \\ z &= \left(\hat{\mu}_t - 1.1\hat{\mu}_{t-1}\right) * \left(\hat{\sigma}_t^2 + 1.1^2 \hat{\sigma}_{t-1}^2\right)^{-\frac{1}{2}} \end{aligned}$$

We test against the one sided null H_0 by rejecting when $z > 1.64$, and since this test has level 0.05 when $\frac{\mu_t}{\mu_{t-1}} = 1.1$ [5], it is even more conservative when $\frac{\mu_t}{\mu_{t-1}} < 1.1$.

4.2.1 Results

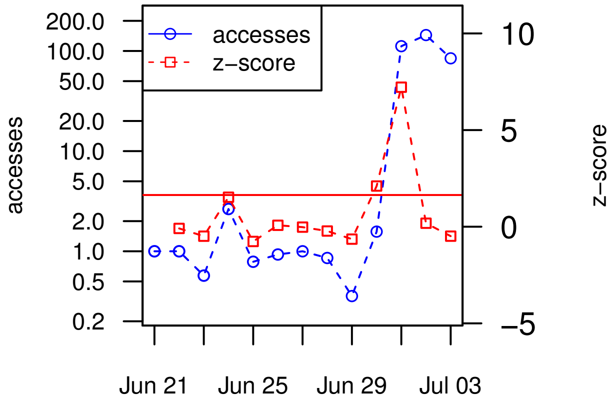

Figure 3: Cache Misses to a particular partition of data in blue, where the value on the first day is normalized to 1. Red shows the z-scores corresponding to the test that the seeks increased less than 10%, and extending above the red line corresponds to rejecting that hypothesis.

In Figures 3 and 4, we display normalized accesses in blue, and the corresponding z-score for a change from the previous day in red. The horizontal red line corresponds to a level 0.05 normal test, and we define our detection algorithm as extending above that line.

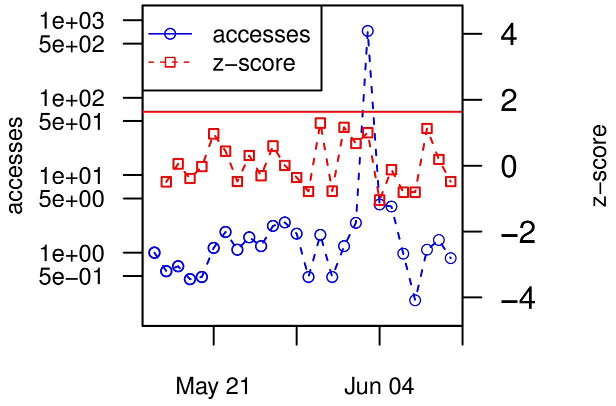

Figure 4: Same as Figure 3, but for a different data partition.

In Figure 3, we see that the cache misses started increasing on June 30th, then ended up 100 times higher on July 1st - 3rd than is was in late June. Our z-score detection algorithm flags this change in behavior on June 30th and July 1st, and this change ended up being a persistent change in behavior of order 100×.

A spike much higher than 100× occurred on June 3rd for the data partition in Figure 4, but this estimate had such high uncertainty that the z-score was moderate, and the change was not flagged. Unlike the case in Figure 3, the higher level did not persist. It is possible that data partitions could see real usage spikes on one day that later disappear and it may be useful to know about those, but after studying the variance, we find that this spike does not present strong evidence of being more than a sampling variation.

5 Conclusion

Many interesting system quantities can be represented as a sum of a vector-valued function over RPCs, and we present a method to obtain estimates of these quantities and their uncertainty from Dapper. At Google, these sampled distributed traces are ubiquitous, and are often the only data source available for some system questions that arise. Although arbitrary traces may have complex sampling structures, we find an unbiased covariance estimate that works in all cases and can be easily computed. We demonstrate how this methodology can be used evaluate to the effectiveness of different bin-packing strategies on Google data centers when evaluating over an extended period, and also for detecting change points in quantities that are not directly logged.

[4]For a given data partition, D, and day t, $f : \lambda \rightarrow x$ would return a binary indicating whether that RPC was a disk seek.

[5]Here we get independence by ignoring the rare traces that span across midnight

A Appendix

Part 1: Notation and Aggregate Representation

We represent all RPCs in our time window of interest, Ω, as a double subscript (i,j), which represents the j^{th} RPC corresponding to trace ID i. This allows us to write

$$\theta = \sum_{i=1}^{N} \sum_{j=1}^{J_i} x_{(i,j)}$$

Letting S be the sample returned by Dapper, $s_{(i,j)}$ and $d_{(i,j)}$ the server sampling and downsampling probabilities for $x_{(i,j)}$, then our estimate can be re-written as

$$\hat{\theta} = \sum_{(i,j)\in S} \frac{x_{(i,j)}}{s_{(i,j)} * d_{(i,j)}}$$

It is useful to re-parametrize the indices (i,j) in terms of the distinct server sampling and downsampling factors:

$$0 < s_1 < s_2 < \ldots < s_M \le 1$$

$$0 < d_1 < d_2 < \ldots < d_L \le 1$$

We then define the (possible empty) index sets as

$$\Omega_{(i,m,l)} = \{(i',j') \in \Omega \quad | \quad i' = i, \quad s_{(i',j')} = s_m \quad d_{(i',j')} = d_l\}$$

the (possibly zero) *trace-level* sums by

$$y_{(i,m,l)} = \sum_{(i,j)\in\Omega_{(i,m,l)}} x_{(i,j)}$$

We define weighted boolean variables

$$W_{(i,m,l)} = \frac{\mathbf{1}_{(i,m,l)\in S}}{s_m d_l}$$

So that

$$T = \sum_{i=1}^{N} \sum_{m=1}^{M} \sum_{l=1}^{L} y_{(i,m,l)} W_{(i,m,l)}$$

Part 2: The Population Covariance Matrix

Before expanding $\mathrm{Cov}(T)$, we note that:

$$\begin{aligned} \mathbb{E}(W_{(i,m,l)}) &= 1 \\ \mathrm{Cov}(W_{(i,m,l)}, W_{(i',m',l')}) &= 0 \qquad \text{If} \qquad i \neq i' \end{aligned}$$

And for $i = i'$, we define $\lambda_{(m,l,m',l')}$ by

$$\begin{aligned} \mathrm{Cov}(W_{(i,m,l)}, W_{(i,m',l')}) &= \frac{\mathbb{P}\Big((i,m,l),(i,m',l') \in S\Big)}{s_m s_{m'} d_l d_{l'}} - 1 \\ &= \frac{(s_m \wedge s_{m'}) * (d_l \wedge d_{l'})}{s_m s_{m'} d_l d_{l'}} - 1 \\ &= \frac{1 - (s_m \vee s_{m'}) * (d_l \vee d_{l'})}{(s_m \vee s_{m'}) * (d_l \vee d_{l'})} \\ &\equiv \lambda_{(m,l,m',l')} \end{aligned}$$

so

$$\mathrm{Cov}(y_{(i,m,l)} W_{(i,m,l)}, y_{(i,m',l')} W_{(i,m',l')}) = \big(y_{(i,m,l)} \otimes y_{(i,m',l')}\big) \lambda_{(m,l,m',l')}$$

Where $\otimes$ is the outer product resulting in a $P \times P$ matrix.

Putting it together, we have

$$\begin{aligned} \Sigma &= \mathrm{Cov}(T) \\ &= \sum_i \sum_{(m,m',l,l')} \big(y_{(i,m,l)} \otimes y_{(i,m',l')}\big) \lambda_{(m,l,m',l')} \end{aligned}$$

Part 3: Unbiased estimate of Σ

Since

$$\mathbb{E}\Big[\frac{\mathbf{1}_{(i,m,l)\in S} \mathbf{1}_{(i',m',l')\in S}}{(s_m \wedge s_{m'}) * (d_l \wedge d_{l'})}\Big] = 1$$

If follow that an unbiased estimate for $y_{(i,m,l)} \otimes y_{(i,m',l')}$ is given by

$$\mathbb{E}\Big[\frac{y_{(i,m,l)} \otimes y_{(i,m',l')} \mathbf{1}_{(i,m,l)\in S} \mathbf{1}_{(i',m',l')\in S}}{(s_m \wedge s_{m'}) * (d_l \wedge d_{l'})}\Big]$$

So an unbiased estimate of Σ is given by

$$\begin{aligned} \hat{\Sigma} &= \sum_{i\in S} \sum_{(m,m',l,l')\in S} \frac{\lambda_{(m,l,m',l')} \big(y_{(i,m,l)} \otimes y_{(i,m',l')}\big)}{(s_m \wedge s_{m'}) * (d_l \wedge d_{l'})} \\ &= \sum_{(m,m',l,l')} \frac{1 - (s_m \vee s_{m'}) * (d_l \vee d_{l'})}{s_m s_{m'} d_l d_{l'}} \sum_{i\in S} y_{(i,m,l)} \otimes y_{(i,m',l')} \end{aligned}$$

Equivalence to GetSigmaHat follows since ProcessSingleTrace produces the above result for a single trace.
A simple case occurs if you are interested in a scalar and each trace shares the same server sampling and downsampling probability. Letting $p_i = s_i * d_i$, the result simplifies to

$$\Sigma = \sigma^2 = \sum_i \frac{1 - p_i}{p_i} y_i^2$$

$$\hat{\Sigma} = \hat{\sigma}^2 = \sum_{i\in S} \frac{1 - p_i}{p_i^2} y_i^2$$

References

[1] Available 20120720: `http://engineering.twitter.com/2012/06/distributed-systems-tracing-with-zipkin.html`.

[2] ALVAREZ, G. A., BOROWSKY, E., GO, S., ROMER, T. H., BECKER-SZENDY, R., GOLDING, R., MERCHANT, A., SPASOJEVIC, M., VEITCH, A., AND WILKES, J. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst. 19*, 4 (Nov. 2001), 483–518.

[3] ANDERSON, E., SPENCE, S., SWAMINATHAN, R., KALLAHALLA, M., AND WANG, Q. Quickly finding near-optimal storage designs. *ACM Trans. Comput. Syst. 23*, 4 (Nov. 2005), 337–374.

[4] BARTH, W. *Nagios: System and Network Monitoring*. No Starch Press, San Francisco, CA, USA, 2006.

[5] BERTOLINO, A., CALABRÒ, A., LONETTI, F., AND SABETTA, A. Glimpse: a generic and flexible monitoring infrastructure. In *Proceedings of the 13th European Workshop on Dependable Computing* (New York, NY, USA, 2011), EWDC '11, ACM, pp. 73–78.

[6] DOWDY, L. W., AND FOSTER, D. V. Comparative models of the file assignment problem. *ACM Comput. Surv. 14*, 2 (June 1982), 287–313.

[7] ERLINGSSON, U., PEINADO, M., PETER, S., AND BUDIU, M. Fay: extensible distributed tracing from kernels to clusters. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 311–326.

[8] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc., 2010.