Overlapping Experiment Infrastructure: More, Better, Faster Experimentation

Diane Tang, Ashish Agarwal, Deirdre O'Brien, Mike Meyer
Google, Inc.
Mountain View, CA
[diane,agarwal,deirdre,mmm]@google.com

ABSTRACT

At Google, experimentation is practically a mantra; we evaluate almost every change that potentially affects what our users experience. Such changes include not only obvious user-visible changes such as modifications to a user interface, but also more subtle changes such as different machine learning algorithms that might affect ranking or content selection. Our insatiable appetite for experimentation has led us to tackle the problems of how to run more experiments, how to run experiments that produce better decisions, and how to run them faster. In this paper, we describe Google's overlapping experiment infrastructure that is a key component to solving these problems. In addition, because an experiment infrastructure alone is insufficient, we also discuss the associated tools and educational processes required to use it effectively. We conclude by describing trends that show the success of this overall experimental environment. While the paper specifically describes the experiment system and experimental processes we have in place at Google, we believe they can be generalized and applied by any entity interested in using experimentation to improve search engines and other web applications.

Categories and Subject Descriptors

G.3 [**Probability and Statistics**]: Experimental Design—*controlled experiments, randomized experiments, A/B testing*; I.2.6 [**Learning**]: [real-time, automation, causality]

General Terms

Measurement, Design, Experimentation, Human Factors, Performance

Keywords

Controlled experiments, A/B testing, Website Testing, MultiVariable Testing

1. INTRODUCTION

Google is a data-driven company, which means that decision-makers in the company want empirical data to drive decisions about whether a change should be launched to users. This data is most commonly gathered by running live traffic experiments. In the context of the web, an experiment consists of a representative segment of traffic (i.e., incoming requests) and a change in what is served to that segment of traffic relative to a control. Both user-visible changes (e.g., changing the background color of the top ads) and non-visible changes, such as testing a new algorithm for predicting the clickthrough rate (CTR) of ads, can be tested via experimentation.

One challenge for supporting this data-driven methodology is keeping up with the rate of innovation. We want to be able to experiment with as many ideas as possible; limiting our rate of change by the number of simultaneous experiments we can run is simply not acceptable. We use experiments to test out new features and to explore the space around existing features. For the latter, experiments are used to learn about user response and optimize what is already running. Imagine that the way we determine what to show on a search results page is parameterized, both in terms of presentation and algorithms. Experiments explore this parameter space by setting different values for the parameters, and we can use the measured impact (with regards to user experience, revenue, and other metrics) to determine where to move in this space to achieve a better result.

While evaluating user response to UI changes is the typical use of experiments, it is worth noting that experimentation is also needed for testing algorithm changes. For example, suppose some team wants to test a new machine learning algorithm for predicting the CTR of ads, or even to test variations of an existing algorithm (e.g, by adjusting the learning or shrinkage rate) [1, 10]. While offline evaluation can be useful for narrowing down the space of options to test, ultimately these options must be tested on live traffic in order to evaluate how well a particular parameterization of an algorithm works in practice (changes may impact user behavior and alter the traffic pattern itself, which would not be caught in offline evaluation). Thus, the evaluation of such machine learning algorithms is limited as much by having the space to experiment as by thinking of alternatives to try.

The design goals for our experiment infrastructure are therefore: *more, better, faster*.

- **More:** We need scalability to run more experiments simultaneously. However, we also need flexibility: different experiments need different configurations and different sizes to be able to measure statistically significant effects. Some experiments only need to change a subset of traffic, say Japanese traffic only, and need to be sized appropriately. Other experiments may change all traffic and produce a large change in metrics, and so can be run on less traffic.
- **Better:** Invalid experiments should not be allowed run on live

KDD'10, July 25–28, 2010, Washington, DC, USA.

traffic. Valid but bad experiments (e.g., buggy or unintentionally producing really poor results) should be caught quickly and disabled. Standardized metrics should be easily available for all experiments so that experiment comparisons are fair: two experimenters should use the same filters to remove robot traffic [7] when calculating a metric such as CTR.

- **Faster:** It should be easy and quick to set up an experiment; easy enough that a non-engineer can do so without writing any code. Metrics should be available quickly so that experiments can be evaluated quickly. Simple iterations should be quick to do. Ideally, the system should not just support experiments, but also controlled ramp-ups, i.e., gradually ramping up a change to all traffic in a systematic and well-understood way.

To meet these design goals, we need not only an experiment infrastructure for running more experiments, but also tools and educational processes to support better and faster experimentation.

For the experiment infrastructure, the obvious solutions are either to have a single layer of experiments or to have multi-factorial experiment design. A single layer means that every query is in at most one experiment, which is easy-to-use and flexible, but simply insufficiently scalable. Multi-factorial experimental design is common in statistical literature [3], where each parameter (factor) can be experimented on independently; each experimental value for a parameter overlaps with every other experiment value for all of the other parameters. Effectively, each query would be in N experiments simultaneously, where N equals the number of parameters. While this approach is backed by years of research and practice, it is impractical in Google's system where we have thousands of parameters that cannot necessarily be varied independently. One simple example is a two parameter system, one for the background color of a web page and another for the text color. While "blue" may be a valid value for both, if both parameters are blue at the same time, then the page will be unreadable.

The solution we propose in this paper is to partition the parameters into subsets, and each subset contains parameters that cannot be varied independently of each other. A subset is associated with a layer that contains experiments, and traffic diversion into experiments in different layers is orthogonal. Each query here would be in N experiments, where N equals the number of layers. While this solution may not sound novel in retrospect, we can find no published papers with this solution.

In this paper, we discuss this layered overlapping experiment infrastructure as well as the associated tools and processes we have put into place to support more, better, and faster experimentation, and show results that support how well we have met those goals. Note that while this paper is specific to Google web search, the more general problem of supporting massive experimentation applies to any company or entity that wants to gather empirical data to evaluate changes. The responses and data gathered may differ, but the approach outlined in this paper should be generalizable.

2. RELATED WORK

Work related to experimentation falls roughly in three areas. The first area is the copious statistical literature on multi-factorial experiments, which we briefly discuss in Section 4.

The second area is the growing body of work on how to run web experiments. Kohavai *et al.* wrote an outstanding paper in KDD that provides both a survey and tutorial on how to run controlled experiments for the web [7]. There are several follow-up papers that describe various pitfalls when running and analyzing these experiments [4, 9]. Krieger has a presentation on how to run A-B tests [8] and Google WebSite Optimizer helps web site designers run their own A-B tests [5]. In general, these papers focus more on how to design and evaluate web-based experiments. The first Kohavi paper is perhaps the most relevant, since several design considerations about how to build an infrastructure for running experiments are discussed, including how to divert traffic into experiments, consistency across queries, interactions, etc. However, none of these papers really addresses the issues involved in scaling an experiment infrastructure and the overall experimentation environment to support running more experiments more quickly and more robustly.

The final area of related work is on interleaved experiments [6], which is focused on a specific type of experiment design (intra-query) used for evaluating ranking changes. This type of experiment may easily be run within the experimental environment that we describe in this paper.

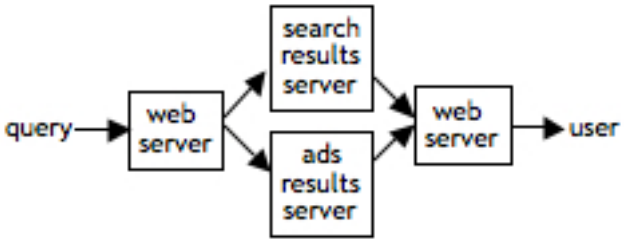

Figure 1: A sample flow of a query through multiple binaries. Information (and time) flows from left to right.

3. BACKGROUND

Before we discuss experimentation at Google, we first describe the environment since it affords opportunities for the design of our infrastructure as well as constraints. Note that in this paper, when we refer to Google or Google's serving infrastructure, we are referring to the web search infrastructure only, and not Google Apps, Android, Chrome, etc.

At a high level, users interact with Google by sending requests for web pages via their browser. For search results pages, the request comes into Google's serving infrastructure and may hit multiple binaries (i.e., programs running on a server machine) before returning the results page to the user. For example, there may be one binary that is responsible for determining which organic search results are most relevant to the query, another for determining which ads are most relevant to the query, and a third for taking both organic and ad results and formatting the resulting web page to return to the user (see Figure 1). On the one hand, this modularization allows us to improve latency (non-dependent processes can run in parallel), clearly separate organic results from ads so that one cannot influence the other, and innovate faster (each binary can evolve separately and is a smaller piece to test allowing for a quicker release cycle). On the other hand, this modularization can require more careful design if every request is allowed to be in at most one experiment. Possible problems include starvation (upstream binaries can starve downstream binaries by allocating all requests to be in experiments prior to the requests being sent downstream) and bias (e.g., an upstream binary may run experiments on all English traffic, leaving downstream binaries with non-English traffic).

Each binary has an associated binary push and data push. The binary push is when new code (bug fixes, performance enhancements, new features, etc.) is incorporated and rolled out to handle live serving; it happens periodically (e.g., weekly). The data push happens more frequently (i.e., on demand or every few hours), and involves pushing updated data to the associated binary. One type

of data included in a data push has to do with the default values for parameters that configure how the binary runs. For example, the binary that controls how results are presented may have a parameter that determines the background color of the top ads block. Another example is a binary that predicts the CTR of ads might have a parameter that controls the rate of learning or shrinkage, i.e., a parameter that controls the step size the algorithm takes during each iteration, impacting both the convergence and local optima chosen. Binaries may have several hundred parameters. New features will likely add one or more parameters: in the simplest case, a single parameter to turn on or off the new feature, and in more complex cases, there may be parameters that govern how the new feature is formatted, numeric thresholds to determine when the new feature is shown, etc. Having separate binary and data pushes means that if we can find the right split, we can take advantage of having both a slow and a fast path for making changes to serving (slow for code, fast for new values for parameters).

An **experiment** in web search diverts some subset of the incoming queries to an alternate processing path and potentially changes what is served to the user. A **control experiment** diverts some subset of incoming queries, but does not change what is served to the user. We use the data push to specify and configure experiments. Thus, in the data push, there is a file (as discussed above) for specifying the default values for the parameters for a binary. There is another file for experiments that specifies how to change what is served to the user by providing alternate values for parameters. Experiments need only specify the parameters that they change; for all other parameters, the default values are used. For example, one simple experiment would change only one parameter, the background color of the top ads, from yellow (the default value) to pink.

In addition to specifying how serving is changed via alternate parameter values, experiments must also specify what subset of traffic is **diverted**. One easy way to do experiment diversion is *random traffic*, which is effectively flipping a coin on every incoming query. One issue with random traffic experiment diversion is that if the experiment is a user-visible change (e.g., changing the background color), the queries from a single user may pop in and pop out of the experiment (e.g., toggle between yellow and pink), which can be disorienting. Thus, a common mechanism used in web experimentation is to use the **cookie** as the basis of diversion; cookies are used by web sites to track unique users. In reality, cookies are machine/browser specific and easily cleared; thus, while a cookie does not correspond to a user, it can be used to provide a consistent user experience over successive queries. For experiment diversion, we do not divert on individual cookies, but rather a **cookie mod**: given a numeric representation of a cookie, take that number modulo 1000, and all cookies whose mod equals 42, for example, would be grouped together for experiment diversion. Assuming cookie assignment is random, any cookie mod should be equivalent to any other cookie mod. Cookie mods are also easy to specify in an experiment configuration and make it easy to detect conflicts: experiment 1 may use cookie mods 1 and 2, while experiment 2 may use cookie mods 3 and 4. Those two experiments would be the same size and, in theory, have comparable traffic.

Configuring experiments in data files makes experiments easy and fast to create: the data files are human readable and easy to edit, they are not code so that experiments can be created by non-engineers, and they are pushed more frequently than code allowing for a "fast path" for experiment creation involving existing parameters only.

Prior to developing our overlapping experiment infrastructure, we used a basic single layer infrastructure. In such an infrastructure, each query is in at most one experiment. Cookie-mod-based experiments were diverted first, followed by random-traffic based experiments. Upstream binaries got "first dibs" on a query, and if the upstream binaries were running enough experiments, then downstream binaries could be starved for traffic to run experiments on. While there were several issues (including having to solve the starvation and bias issues mentioned above), this single layer infrastructure did meet a few of our design goals: it was easy to use and reasonably flexible. However, given Google's data-driven culture, the single layer approach is not sufficiently scalable: we cannot run enough experiments fast enough.

4. OVERLAPPING EXPERIMENT INFRASTRUCTURE

In this section, we describe the overlapping experiment infrastructure, which tries to keep the advantages of the single layer system (ease of use, speed) while increasing scalability, flexibility and robustness. We also enable the gradual ramping-up of launches in a controlled, well-defined fashion.

The obvious statistical solution is a multi-factorial system, where each factor corresponds to a changeable parameter in the system. Effectively, a request would be in N simultaneous experiments, where each experiment would modify a different parameter and N equals the number of parameters. Multi-factorial experiments are backed by copious theory and practice [3]. However, a multi-factorial system is simply not feasible in our complex environment, since not all parameters are independent and not all values that we may want to test for a parameter work with the values for another parameter (e.g., pink text color on a pink background). In other words, Google has to always serve a readable, working web page.

Given this constraint, our main idea is to partition parameters into N subsets. Each subset is associated with a layer of experiments. Each request would be in at most N experiments simultaneously (one experiment per layer). Each experiment can only modify parameters associated with its layer (i.e., in that subset), and the same parameter cannot be associated with multiple layers.

The obvious question is how to partition the parameters. First, we can leverage the modularization into multiple binaries: parameters from different binaries can be in different subsets (which solves the starvation and bias issues mentioned above). However, all parameters for a given binary do not need to be in a single subset: we can further partition the parameters within a binary either by examination (i.e., understanding which parameters cannot be varied independently of one another) or by examining past experiments (i.e., empirically seeing which parameters were modified together in previous experiments). Looking at Figure 1, we could have one or more layers each for the web server, the server for organic search results, and the server for ads.

In fact, the system that we designed is more flexible than simply partitioning the parameters into subsets that are then associated with layers. To explain the flexibility, we introduce several definitions. Working within the space of incoming traffic and the system parameters, we have three key concepts:

- A **domain** is a segmentation of traffic.
- A **layer** corresponds to a subset of the system parameters.
- An **experiment** is a segmentation of traffic where zero or more system parameters can be given alternate values that change how the incoming request is processed.

We can nest domains and layers. Domains contain layers. Layers contain experiments, and can also contain domains. Nesting a domain within a layer allows for the subset of parameters associated

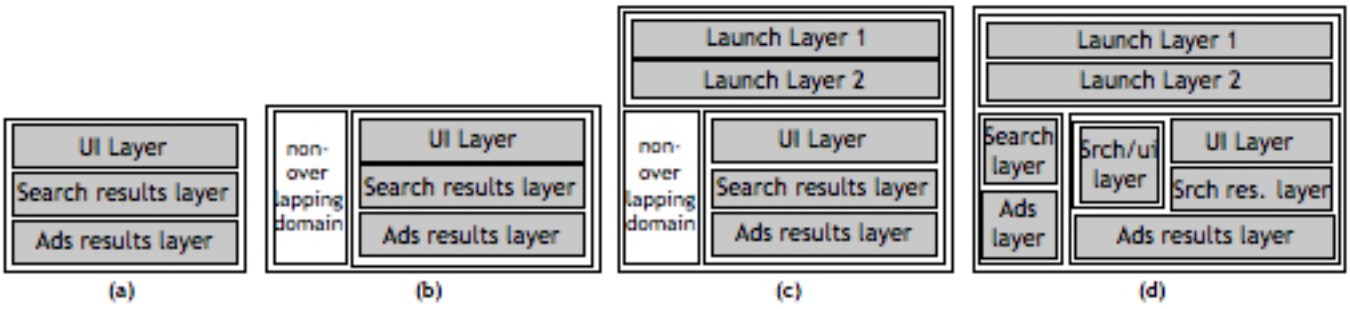

Figure 2: A diagram of (a) basic overlapping set-up with three layers and (b) a set-up with both non-overlapping and overlapping domains, (c) a set-up with non-overlapping, overlapping, and launch domains, and (d) a complex set-up with multiple domains. An incoming request would correspond to a vertical slice through the configuration; i.e., an incoming request in (b) would either be in a single experiment (in the non-overlapping domain) or in at most three experiments, one each for the UI layer, search results layer, and ads results layer.

with the layer to be partitioned further within that nested domain. To get us started, we have the default domain and layer that contain both all traffic and all parameters. Within the default domain and layer, we could, for example:

- Simply segment the parameters into three layers (Figure 2a). In this case, each request would be in at most three experiments simultaneously, one for each layer. Each experiment could only modify the parameters corresponding to that layer.
- First segment traffic into two domains. One domain could be a domain with a single layer (the non-overlapping domain), and the other domain would be the overlapping domain with three layers (Figure 2b). In this case, each request would first be assigned to either the non-overlapping or overlapping domain. If the request was in the non-overlapping domain, then the request would be in at most one experiment (and could change any parameter in the entire space of parameters). If the request was in the overlapping domain, then the request would be in at most three experiments, one for each layer, and each experiment could only use the parameters corresponding to that layer.

While this nesting may seem complex, it affords several advantages. First, having a non-overlapping domain allows us to run experiments that really need to change a wide swath of parameters that might not normally be used together. Next, it allows us to have different partitionings of parameters; one could imagine three domains, one non-overlapping, one overlapping with one partitioning of the parameters, and a third overlapping domain with a different parameter partitioning. Finally, the nesting allows us to more efficiently use space, depending on which partitionings are most commonly used, and which cross-layer parameter experiments are most commonly needed. Note that it is easy to move currently unused parameters from one layer to another layer, as long as one checks to make sure that the parameters can safely overlap with the parameters in the original layer assignment [1]. Also note that to ensure that the experiments in different layers are independently diverted, for cookie-mod based experiments, instead of mod = f(cookie) % 1000, we use mod = f(cookie, layer) % 1000. While this nesting complexity does increase flexibility, there is a cost to changing the configuration, especially of domains: changing how traffic is allocated to domains changes what traffic is available to experiments. For example, if we change the non-overlapping domain from 10% of cookie mods to 15%, the additional 5% of cookie mods comes from the overlapping domain and cookies that were seeing experiments from each layer in the the overlapping domain are now seeing experiments from the non-overlapping domain.

An additional concept is that of **launch layers**. Launch layers differ from the experiment layers discussed up to this point in several key ways:

- Launch layers are always contained within the default domain (i.e., they run over all traffic).
- Launch layers are a separate partitioning of the parameters, i.e., a parameter can be in at most one launch layer and at most one "normal" layer (within a domain) simultaneously.
- In order to make this overlap of parameters between launch and normal layers work, experiments within launch layers have slightly different semantics. Specifically, experiments in launch layers provide an alternative default value for parameters. In other words, if no experiments in the normal experiment layers override a parameter, then in the launch layer experiment, the alternate default value specified is used and the launch layer experiment behaves just like a normal experiment. However, if an experiment in the normal experiment layer does override this parameter, then that experiment overrides the parameter's default value, regardless of whether that value is specified as the system default value or in the launch layer experiment.

Examples of launch layers are shown in Figure 2c,d. Defining launch layers in this way allows us to gradually roll out changes to all users without interfering with existing experiments and to keep track of these roll-outs in a standardized way. The general usage of launch layers is to create a new launch layer for each launched feature and to delete that layer when the feature is fully rolled out (and the new parameter values are rolled into the defaults). Finally, because experiments in launch layers are generally larger, they can be used to test for interactions between features. While in theory we can test for interactions in the normal experiment layers (assuming that we either set up the experiments manually if the parameters are in the same layer or look at the intersection if the parameters are in different layers), because experiments are smaller in the normal layers, the intersection is smaller and therefore interactions are harder to detect.

[1]Sociologically, we have observed that if layers have semantically meaningful names, e.g., the "Ad Results Layer" and the "Search Results Layer", engineers tend to be reluctant to move flags that violate that semantic meaning. Meaningful names can help with robustness by making it more obvious when an experiment configuration is incorrect, but it can also limit the flexibility that engineers will take advantage of.

Recall that both experiments and domains operate on a segment of traffic (we call this traffic the "diverted" traffic). Diversion types and conditions are two concepts that we use in order to determine what that diverted segment of traffic is.

We have already described two **diversion types** earlier in Section 3, namely cookie-mods and random traffic. Also discussed above is how cookie mod diversion changes with layers to also incorporate the layer id (mod = f(cookie, layer) % 1000) to ensure orthogonality between layers. Two other diversion types that we support are user-id mods and cookie-day mods. User-id mods are like cookie mods, except that we use the signed-in user id instead of the cookie. For cookie-day mods, we take the mod of the cookie combined with the day, so that for a given day, a cookie is in an experiment or not, but the set of cookies in an experiment changes from day-to-day. In all cases, there is no way to configure an experiment so that a specific cookie or user gets diverted to it. Similarly, analyses always use aggregates over groups of queries, cookies, or users. Also note that while we currently support four diversion types, we could support other diversion types, e.g., by hashing the query string.

The main reasons for different diversion types are to ensure consistency across successive queries and to potentially uncover any learning effects over time. Given these reasons, we divert traffic by diversion type in a particular order: user id, cookie, cookie-day, and finally random traffic. Once an event meets the criteria for a particular experiment in one diversion type, it is not considered by the remaining diversion types (see Figure 3). While this order ensures maximal consistency, the one downside is, for example, that a 1% random traffic experiment gets fewer requests than a 1% cookie mod experiment in the same layer. At the extreme, we can see the same starvation effect that we used to see between upstream and downstream binaries. In practice, layers tend to have a predominant diversion type, and experiments and controls must have the same diversion type. The main impact is that different diversion types require different experiment sizes (see Section 5.2.1).

After selecting a subset of traffic by diversion type, **conditions** provide better utilization of this traffic by only assigning specific events to an experiment or domain. For example, an experiment that only changes what gets served in queries coming from Japan may include a "Japan" condition. We support conditions based on country, language, browser, etc. With conditions, an experiment that only needs Japanese traffic can use the same cookie mod as another experiment that only needs English traffic, for example. Another use of conditions is to **canary** new code (the code itself is pushed via a binary push), i.e., test new code on a small amount of traffic and make sure that the new code is not buggy and works as expected before running it on more traffic (the canary is checked for bugs via error logs from the binaries and real-time monitoring of metrics). To support this use case, we provide conditions based on machine or datacenter to further restrict traffic to an experiment. While canary experiments do not replace rigorous testing, they are a useful supplement, since it both limits the potential damage while subjecting the new code to the variety of requests in live traffic that is hard to duplicate in a test.

Conditions are specified directly in the experiment (or domain) configuration, allowing us to do conflict detection based on the data files at experiment creation time. As mentioned in the diversion type section, once an event meets the *diversion-type* criteria for an experiment in one diversion type, it is not considered by the remaining diversion types even if it does not meet the conditions for assignment to an experiment in the earlier diversion type. The importance of this is perhaps best explained by example. If we take all traffic corresponding to a particular cookie mod, we have an unbiased diversion. But consider the case of two experiments on a given cookie mod – one conditioned on Japanese traffic, the other conditioned on English traffic. The rest of the traffic (traffic in languages other than Japanese and English) for the same cookie mod will not be assigned to any experiment in the cookie diversion type. To avoid **biased** traffic in subsequent diversion types, it is important that this otherwise available traffic (events meeting the diversion-type criteria but not meeting any experiment conditions) not be assigned to experiments in subsequent diversion types. We avoid this by tagging this unassigned traffic with a biased id.

Figure 3 shows the logic for determining which domains, layers, and experiments a request is diverted into. All of this logic is implemented in a shared library compiled into the binaries, so that any changes (e.g., new types of conditions, new diversion types, etc.) are incorporated into all binaries during their regular binary pushes. Given the complexity of the implementation, a shared library allows for a consistent implementation across binaries and means that new functionality automatically gets shared.

Given this infrastructure, the process of evaluating and launching a typical feature might be something like:

- Implement the new feature in the appropriate binary (including code review, binary push, setting the default values, etc. as per standard engineering practices).
- Create a canary experiment (pushed via a data push) to ensure that the feature is working properly. If not, then more code may need to be written.
- Create an experiment or set of experiments (pushed via a data push) to evaluate the feature. Note that configuring experiments involve specifying the diversion type and associated diversion parameters (e.g., cookie mods), conditions, and the affected system parameters.
- Evaluate the metrics from the experiment. Depending on the results, additional iteration may be required, either by modifying or creating new experiments, or even potentially by adding new code to change the feature more fundamentally.
- If the feature is deemed launchable, go through the launch process: create a new launch layer and launch layer experiment, gradually ramp up the launch layer experiment, and then finally delete the launch layer and change the default values of the relevant parameters to the values set in the launch layer experiment.

5. TOOLS & PROCESSES

While having the overlapping infrastructure is necessary to be able to scale to running more experiments faster and evalute more changes concurrently, the infrastructure by itself is not sufficient. We also need tools, research, and educational processes to support the faster rate of experimentation. In this section, we discuss several key tools and processes, and how they have helped us scale.

5.1 Tools

Data File Checks: One advantage of data files is that they can be automatically checked for errors, which leads to fewer broken experiments being run. We have automated checks for syntax errors (are all the required fields there and parseable), consistency and constraints errors (i.e., uniqueness of id's, whether the experiment is in the right layer given the parameters used, whether the layer has enough traffic to support the experiment, traffic constraint checks: is the experiment asking for traffic already claimed by another experiment, etc.; note that these checks can get tricky as the set of possible diversion conditions grows), and basic experiment design checks (does the experiment have a control, is the control in the

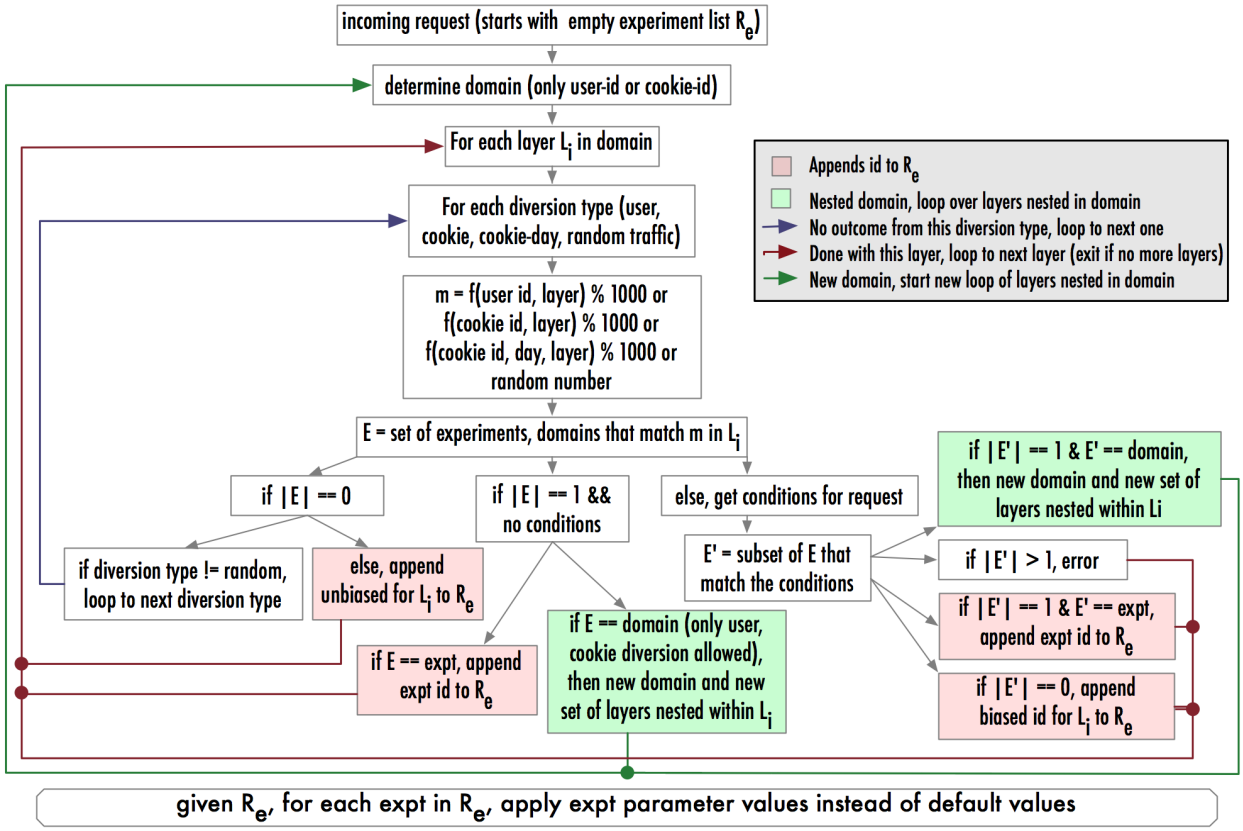

Figure 3: Logic flow for determining which domains, layers, and experiments a query request is in.

same layer as the experiment, does the control divert on the same set of traffic as the experiment, etc.).

Real-time Monitoring: We use real-time monitoring to capture basic metrics (e.g., CTR) as quickly as possible in order to determine if there is something unexpected happening. Experimenters can set the expected range of values for the monitored metrics (there are default ranges as well), and if the metrics are outside the expected range, then an automated alert is fired. Experimenters can then adjust the expected ranges, turn off their experiment, or adjust the parameter values for their experiment. While real-time monitoring does not replace careful testing and reviewing, it does allow experimenters to be aggressive about testing potential changes, since mistakes and unexpected impacts are caught quickly.

5.2 Experiment Design & Sizing

Experiment design and sizing go beyond the basic checks performed on data files (e.g., that every experiment must have a control that uses the same diversion conditions).

5.2.1 Sizing

As Kohavi mentions [7], experiments should be sized to have enough statistical power to detect as small a metric change as considered interesting or actionable. In this section, we discuss both how to size experiments and the dependency on experiment set-up, as well as an associated experiment sizing tool.

Define the effective size of an experiment as:

$$N = (1/queries_{control} + 1/queries_{experiment})^{-1}$$

. In practice we are interested in the individual terms $queries_{control}$ and $queries_{experiment}$, but it is through N that these affect the variance of the relative metric estimates. To determine N correctly, we need to know:

- Which metric(s) the experimenter cares about,
- For each metric, what change the experimenter would like to detect (θ), e.g., the experimenter wants to be able to detect a 2% change in click through rate,
- For each metric, the standard error for a one unit (i.e. $N = 1$) sample (s). Thus the standard error for an experiment of size N is $s/\sqrt{N}$.

Kohavi assumes that the experiment and control are the same size, i.e., $queries_{experiment} = 2N$ and so must be greater than or equal to $16(s/\theta)^2$ to meet the detection requirement. The number 16 is determined both by the desired confidence level ($1-\alpha$, often 95%) and desired statistical power ($1-\beta$, often 80%).

One advantage of our overlapping set-up is that we can create a large control in each layer that can be shared among multiple experiments. If the shared control is much larger than the experiment ($1/queries_{control} + 1/queries_{experiment} \approx 1/queries_{experiment}$), then we can use $queries_{experiment} = N$ rather than $2N$, leading to a smaller experiment size of $queries_{experiment} = N \geq 10.5(s/\theta)^2$ while gaining statistical power ($1-\beta$) of 90% [2].

The bigger issue we encountered in sizing experiments is how to estimate s, the standard error, especially since we use many ratio metrics, y/z (e.g., coverage, the percentage of queries on which we show ads (queries with an ad / total queries)). The problem arises when the unit of analysis is different than the experimental unit. For example, for coverage, the unit of analysis is a query, but

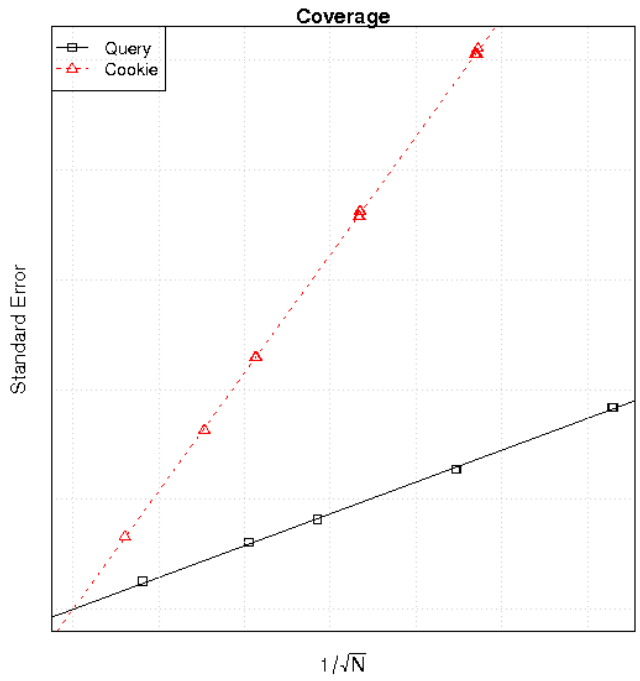

Figure 4: Slope for calculating s for coverage by diversion type.

for cookie-mod experiments, the experimental unit is a cookie (a sequence of queries) and we cannot assume that queries from the same user or cookie are independent. Our approach is to calculate $s\prime$, the standard error per experimental unit and then write s in terms of $s\prime$, e.g. here $s\prime$ would be the standard error per cookie-mod, and $s = s\prime\sqrt{avg\ queries\ per\ cookie\text{–}mod}$. For ratio metrics, we calculate $s\prime$ using the delta method [11].

Figure 4 shows the standard error against $1/\sqrt{N}$ for the coverage metric in different experiments—both cookie-mod and random traffic experiments. The slope of the line gives s. While the axis labels are elided for confidentiality, it is apparent that the slope of the cookie line is much steeper than the slope of the query line, i.e., to measure the same change in coverage with the same precision, a cookie mod experiment will need to be larger than the corresponding random traffic experiment.

Since s differs by both metric and by diversion type, rather than having experimenters calculate these values individually, we provide a sizing tool. Experimenters specify the metric(s) and the change they want to be able to detect, the diversion type (e.g., cookie-mod vs. random traffic), and what segment of traffic they are diverting on (e.g., the conditions, such as Japanese traffic only). This tool then tells the experimenter how much traffic their experiment will need to detect the desired change with statistical significance. Experimenters can easily explore the trade-offs with regards to what size change can be detected with what amount of traffic. With a single canonical tool, we gain confidence that experiments will be sized properly before they are run.

To gather data for our sizing tool, we constantly run a set of **uniformity trials**, i.e., we run many controls or A vs. A experiments, varying both experiment size and diversion type. We can use the results to empirically measure the natural variance of our metrics and test the accuracy of our calculated confidence intervals.

5.2.2 *Triggering, Logging, & Counter-factuals*

As a reminder, diversion refers to the segment of traffic in an experiment. However, an experiment may not actually change serving on all diverted requests. Instead, the experiment may **trigger** only on a subset of the diverted requests. For example, an experiment that is testing when to show weather information on a query may get all traffic diverted to it, but only show the weather information on a subset of those queries; that subset is called the trigger set.

Often, we cannot divert only on the trigger set since determining which requests would trigger the change may require additional processing at runtime; this need for additional processing is why triggers cannot be implemented as conditions (the information is available too late in the control flow). Thus, it is important to log both the factual (when the experiment triggered) and the **counter-factual** (when the experiment would have triggered). The counter-factual is logged in the control. For example, in the experiment mentioned above, the factual (when the weather information is shown) is logged in the experiment, while the counter-factual is logged in the control, i.e., when the weather information would have been shown on this query (but was not since this is the control). This logging is important for both sizing the experiment and analyzing the experiment, since including the unchanged requests dilutes the measured impact of the experiment. Restricting to the trigger set allows experimenters to measure their impact more accurately. In addition, by focusing on the larger effect in the trigger set, the amount of traffic needed is reduced since the effective size of the experiment depends on the inverse of the square of the effect that we aim to detect ($1/\theta^2$).

5.2.3 *Pre- & post-periods*

A pre-period is a period of time prior to starting the experiment where the same traffic (i.e., the same cookie mods) is diverted into the experiment but no changes are made to serving. A post-period is the same thing, but after the experiment. Both periods are akin to comparing a control to a control but using the traffic that is actually diverted to the experiment. Pre-periods are useful for ensuring that the traffic diverted into an experiment really is comparable to its control and does not have any issues, for example with uncaught spam or robots. Post-periods are useful for determining if there are any learned effects from running the experiment. These techniques only apply to user-id mod and cookie mod experiments.

5.3 Fast Analytics

While the infrastructure and tools mentioned so far enable many simultaneous experiments and expedite running an experiment, the actual experimentation process will not be substantially faster unless experiment analysis is also addressed. A full discussion of experimental analysis tools is beyond the scope of this paper, but we briefly discuss the main design goals here.

The primary goal of the analysis tool is to provide accurate values for the suite of metrics that experimenters examine to evaluate their experiment. At Google, rather than combining multiple metrics into a single objective function, we examine a suite of metrics to more fully understand how the user experience might change (e.g., how quickly the user can parse the page, how clicks might move around, etc.). Note that live traffic experiments can only measure what happens and not why these changes happen.

Beyond accuracy and completeness, other key design goals for an experiment analysis tool include:

- Correctly computed and displayed confidence intervals: experimenters need to understand whether the experiment simply has not received enough traffic (confidence intervals are too wide) or whether the observed changes are statistically significant. We have researched a number of ways of calculating accurate confidence intervals and while a full discussion is beyond the

scope of this paper, we note that we have considered both delta method approaches (as mentioned previously) and an alternate empirical method for computing confidence intervals: carve the experiment up into smaller subsets and estimate the variance from those subsets. Also note that care must be taken in looking at multiple metrics and experiments, since if enough are examined, some value will randomly be shown as significant.

- A good UI: it needs to be easy to use and easy to understand. Graphs are useful; even simple graphs like sparklines can help visualize if the aggregate change is consistent over time periods. The UI should also point out when improper comparisons are made (e.g., comparing experiments across layers), and make it easy to change which experiments are compared, the time period, etc.
- Support for slicing: aggregate numbers can often be misleading, as a change may not be due to the metric actually changing (e.g., CTR changing), but may rather be due to a mix shift (e.g., more commercial queries). These Simpson's paradoxes are important to spot and understand, as Kohavi mentions [4].
- Extensibility: it must be easy to add custom metrics and slicings. Especially for new features, the existing suite of metrics and slicings may be insufficient.

Having a single tool to provide accurate metrics for experiments means that we have a single consistent implementation, with agreed upon filters (e.g., to remove potential robot traffic or spam), so that different teams that measure CTR know that their values are comparable. A single tool is also more efficient, since the computation is done once and presented to the users, rather than each experimenter running their own computations.

5.4 Education

While the overlapping infrastructure and accompanying tools and experiment design address the technical requirements to enable more, better, faster experimentation, we also need to address the people-side. Education is equally necessary to facilitate robust experimentation. At Google, two processes have helped to ensure that experiments are well-designed and that the results of an experiment are understood and disseminated.

5.4.1 Experiment Council

The first process is something that we call experiment council, which consists of a group of engineers who review a light-weight checklist that experimenters fill out prior to running their experiment. Questions include:

- basic experiment characterization (e.g., what does the experiment test, what are the hypotheses),
- experiment set-up (e.g., which experiment parameters are varied, what each experiment or set of experiments tests, which layer),
- experiment diversion and triggering (e.g., what diversion type and which conditions to use for diversion, what proportion of diverted traffic triggers the experiment),
- experiment analysis (e.g., which metrics are of interest, how big of a change the experimenter would like to detect),
- experiment sizing and duration (to ensure that, given the affected traffic, the experiment has sufficient statistical power to detect the desired metric changes),
- experiment design (e.g., whether a pre- or post-period is warranted, whether counter-factual logging is correctly done, etc.).

First-time experimenters learn about proper experiment design and sizing as well as the technical details behind implementing the experiment. Repeat experimenters find the checklist light-weight enough to still be useful. Moreover, the process is a useful way for disseminating updated best practices with regards to experimentation (e.g., pointers to new tools to facilitate experimentation, new metrics that may be useful, etc.). The checklist is hosted on a web application, which is useful both for archival purposes as well as educational: new experimenters can read past checklists to understand the issues.

5.4.2 Interpreting the Data

The other process we put in place is a forum where experimenters bring their experiment results to discuss with experts. The goal of the discussion is to:

- Ensure that the experiment results are valid. There are times, even with experiment council, where something in the actual implementation goes awry, or something unexpected happens. In those cases, the discussion is as much a debugging session as anything else. Having experts familiar with the entire stack of binaries, logging, experiment infrastructure, metrics, and analytical tools is key.
- Given valid results, make sure that the metrics being looked at are a complete set with regards to understanding what is happening. Other ways of slicing the data or metric variations may be suggested to gain a more complete understanding of the impact of an experiment. Some experiments are complex enough that experimenters come multiple times with follow-ups.
- Given a full set of results, discuss and agree on whether overall the experiment is a positive or negative user experience, so that decision-makers can use this data (combined with other strategic or tactical information) to determine whether to launch this change, suggest possible refinements, or give up.

The discussion forum is useful for experimenters to better learn how to interpret experiment results. Repeat experimenters generally do not make the same mistakes and can anticipate what analysis needs to be done to gain a full understanding. The discussion forum is also open, so that future experimenters can attend in order to learn in advance of running their experiment. Experiments are also documented so that we have a knowledge repository.

6. RESULTS

We first deployed our overlapping experiment infrastructure in March 2007 (various tools and processes pre-dated or post-dated the infrastructure launch). Ultimately, the success of this overall system is measured in how well we met our goals of running more experiments, running them better, and getting results faster.

6.1 More

We can use several measures to determine if we are successfully running more experiments: how many experiments are run over a period of time, how many launches resulted from those experiments, and how many different people run experiments (see Figure 5). For the number of experiments, note that we are including control experiments in the count. For the count of unique entities, some experiments have multiple owners (e.g., in case anyone is out of town and something happens) or team mailing lists included as owners, both of which are included in the count. Unfortunately, we do not have an easy way to determine how many owners are non-engineers, but anecdotally, this number has increased over time. For the number of launches, we only present the numbers after the launch of overlapping experiments. Prior to overlapping experiments, we used other mechanisms to launch changes; after overlap-

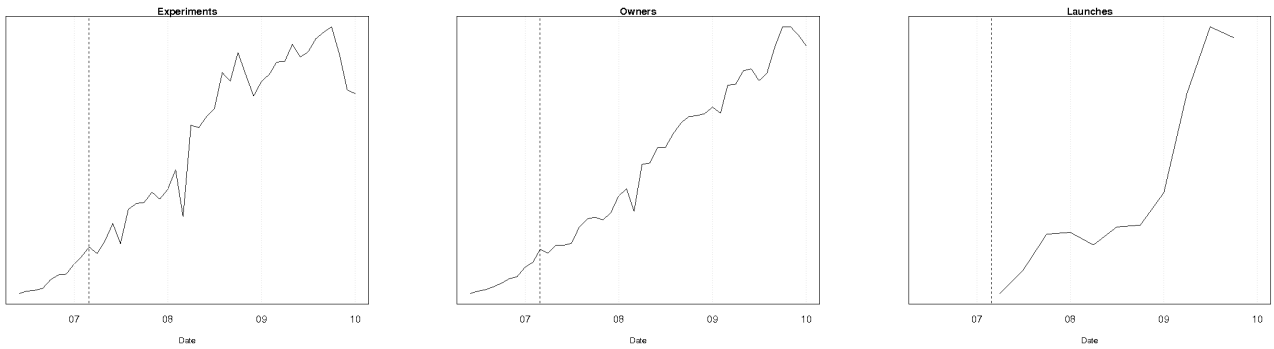

Figure 5: Graphs showing the trend over time for the number of experiments, people running experiments, and launches.

ping experiments, we still use other mechanisms to launch changes, but with decreasing frequency. The y-axes on all of these graphs have been elided for confidentiality (they are on a linear scale), but the trends are clear in showing that we have enabled nearly an order of magnitude more experiments, more launches, and more people running experiments with this overall system.

6.2 Better

Another measure for the success of our overall system, tools, and education is whether the experiments we run now are better than before. We only have anecdotal data here, but we are members of experiment council and the discussion forum and have seen many of the experiments before and after this system was deployed. Our observations are that we see:

- Fewer misconfigured experiments, although we do still encounter the occasional logging issue (for counter-factuals) or weird error / failure cases.
- Fewer forgotten experiments (i.e., people who start experiments and then forget to analyze them).
- Fewer discussions about "what exactly are you measuring here for CTR" or "what filters are you using"; with a canonical analysis tool, the discussion can focus now solely on the interpretation of the metrics rather than on making sure that the definition and calculation of the metrics makes complete sense.
- Better sanity checks, e.g., with pre-periods, to ensure that there are no issues with the traffic being sent to the experiment.

While ideally "fewer" would actually be "none", overall it seems that there are fewer mistakes and problems being made in the design, implementation, and analysis of experiments despite the fact that we have even more people running even more experiments.

6.3 Faster

A final measure of the success of our overall system is whether we are able to ultimately get data faster and make decisions more quickly. For speed, we again do not have empirical data, but we can discuss the general perception of the speed of experimentation here. Experimentation can be broken up into several phases:

- Implementing a new feature to test it out. This phase is now the slowest part, and so we have built other tools (beyond the scope of this paper) to expedite building and testing prototypes (i.e., separating out the process of building something for experimentation purposes vs. building the production-ready version).
- Pushing a new experiment to serving given an implemented feature. This phase can take minutes to hours to create depending on the complexity of the parameters, a negligible amount of time to run the pre-submit checks (seconds to minutes), and then a time comparable to creation time to review. The time needed for the data push depends on the binary, but ranges from 1-2 hours (which includes time running a canary) to half a day.
- Running the experiment depends on the sizing and how long it takes to get statistically significant numbers. We can typically get at least a feel for what is happening with some basic metrics within a few hours after the experiment starts running. The total duration depends, both on the number of iterations the experiment needs, the sizing, the urgency, etc.
- Analyzing the experiment is also variable. In many cases, no custom analysis is needed at all, or only slight extensions to the analysis tool. In those cases, the analysis can often be reasonably quick (days). However, in other cases, custom analysis is still needed in which case the analysis time is highly variable.

Overall, the current pain points include the time needed to implement an experiment, some time waiting for a pre-period to run, and in custom analysis. Those are all areas we are still working on.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have described the design of an overlapping experiment infrastructure and associated tools and educational processes to faciliate more experimentation, better and more robust experimentation, and faster experimentation. We have also given results that show the practical impact of this work: more experiments, more experimenters, more launches, all faster and with fewer errors. While the actual implementation is specific to Google, the discussion of the design choices throughout the paper should be generalizable to any any entity that wants to gather empirical data to evaluate changes.

There are several areas in which we are continuing to improve our experiment infrastructure, including:

- Expediting the implementation of a new feature and facilitating radically different experiments (beyond what can be expressed via parameters).
- Pushing on the limit of whether an experiment parameter is really limited to a single layer. Especially for numeric parameters, we have added operators (e.g., multiplication, addition) that are transitive and therefore composable. With these operators, we can use the same parameter in experiments in multiple layers, as long as those experiments only specify operations on the default value rather than overriding the default value.
- There are times when we need to run experiments that focus on small traffic slices, such as rare languages (e.g., Uzbek or Swahili). Given the sheer volume of experiments that we run,

it is often difficult to carve out the space to run a sufficiently large experiment to get statistically significant results within a reasonable time frame.

- Continuing to push on efficient experiment space utilization by providing even more expressive conditions (and the associated verification to ensure robustness), etc.

We continue to innovate on experimentation since the appetite for experimentation and data-driven decisions keeps growing.

Acknowledgments: Many folks beyond the authors participated in the work described here. An incomplete list includes Eric Bauer, Ilia Mirkin, Jim Morrison, Susan Shannon, Daryl Pregibon, Diane Lambert, Patrick Riley, Bill Heavlin, Nick Chamandy, Wael Salloum, Jeremy Shute, David Agraz, Simon Favreau-Lessard, Amir Najmi, Everett Wetchler, Martin Reichelt, Jay Crim, and Eric Flatt. Thanks also to Robin Jeffries, Rehan Khan, Ramakrishnan Srikant, and Roberto Bayardo for useful comments on the paper.

8. REFERENCES

[1] D. Agarwal, A. Broder, D. Chakrabarti, D. Diklic, V. Josifovski, and M. Sayyadian. Estimating rates of rare events at multiple resolutions. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*, 2007.

[2] W. G. Cochran. *Sampling Techniques*. Wiley, 1977.

[3] D. Cox and N. Reid. The theory of the design of experiments, 2000.

[4] T. Crook, B. Frasca, R. Kohavi, and R. Longbotham. Seven pitfalls to avoid when running controlled experiments on the web. Microsoft white paper, March 2008. http://exp-platform.com/Documents/ExPpitfalls.pdf.

[5] Google. Google website optimizer. http://www.google.com/analytics/siteopt.

[6] T. Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*, 2002.

[7] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne. Controlled experiments on the web: Survey and practical guide. *Data Mining and Knowledge Discovery*, 18, no. 1:140–181, July 2008.

[8] M. Krieger. Wrap up & experimentation: Cs147l lecture, 12 2009. http://hci.stanford.edu/courses/cs147/lab/slides/08-experimentation.pdf.

[9] Microsoft. Microsoft's experimentation platform. http://exp-platform.com/default.aspx.

[10] M. Richardson, E. Dominowska, and R. Ragno. Predicting clicks: Estimating the click-through rate for new ads. In *Proceedings of the 16th International World Wide Web Conference*, 2007.

[11] L. Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer Texts, 2004.