Scaling Big Data Mining Infrastructure: The Twitter Experience

Jimmy Lin and Dmitriy Ryaboy
Twitter, Inc.
@lintool @squarecog

ABSTRACT

The analytics platform at Twitter has experienced tremendous growth over the past few years in terms of size, complexity, number of users, and variety of use cases. In this paper, we discuss the evolution of our infrastructure and the development of capabilities for data mining on "big data". One important lesson is that successful big data mining in practice is about much more than what most academics would consider data mining: life "in the trenches" is occupied by much preparatory work that precedes the application of data mining algorithms and followed by substantial effort to turn preliminary models into robust solutions. In this context, we discuss two topics: First, schemas play an important role in helping data scientists understand petabyte-scale data stores, but they're insufficient to provide an overall "big picture" of the data available to generate insights. Second, we observe that a major challenge in building data analytics platforms stems from the heterogeneity of the various components that must be integrated together into production workflows—we refer to this as "plumbing". This paper has two goals: For practitioners, we hope to share our experiences to flatten bumps in the road for those who come after us. For academic researchers, we hope to provide a broader context for data mining in production environments, pointing out opportunities for future work.

1. INTRODUCTION

The analytics platform at Twitter has experienced tremendous growth over the past few years in terms of size, complexity, number of users, and variety of use cases. In 2010, there were approximately 100 employees in the entire company and the analytics team consisted of four people—the only people to use our 30-node Hadoop cluster on a daily basis. Today, the company has over one thousand employees. There are thousands of Hadoop nodes across multiple datacenters. Each day, around one hundred terabytes of raw data are ingested into our main Hadoop data warehouse; engineers and data scientists from dozens of teams run tens of thousands of Hadoop jobs collectively. These jobs accomplish everything from data cleaning to simple aggregations and report generation to building data-powered products to training machine-learned models for promoted products, spam detection, follower recommendation, and much, much more. We've come a long way, and in this paper, we share experiences in scaling Twitter's analytics infrastructure over the past few years. Our hope is to contribute to a set of emerging "best practices" for building big data analytics platforms for data mining from a case study perspective.

A little about our backgrounds: The first author is an Associate Professor at the University of Maryland who spent an extended sabbatical from 2010 to 2012 at Twitter, primarily working on relevance algorithms and analytics infrastructure. The second author joined Twitter in early 2010 and was first a tech lead, then the engineering manager of the analytics infrastructure team. Together, we hope to provide a blend of the academic and industrial perspectives—a bit of ivory tower musings mixed with "in the trenches" practical advice. Although this paper describes the path we have taken at Twitter and is only one case study, we believe our recommendations align with industry consensus on how to approach a particular set of big data challenges.

The biggest lesson we wish to share with the community is that successful big data mining is about much more than what most academics would consider data mining. A significant amount of tooling and infrastructure is required to operationalize vague strategic directives into concrete, solvable problems with clearly-defined metrics of success. A data scientist spends a significant amount of effort performing exploratory data analysis to even figure out "what's there"; this includes data cleaning and data munging not directly related to the problem at hand. The data infrastructure engineers work to make sure that productionized workflows operate smoothly, efficiently, and robustly, reporting errors and alerting responsible parties as necessary. The "core" of what academic researchers think of as data mining—translating domain insight into features and training models for various tasks—is a comparatively small, albeit critical, part of the overall insight-generation lifecycle.

In this context, we discuss two topics: First, with a certain amount of bemused ennui, we explain that schemas play an important role in helping data scientists understand petabyte-scale data stores, but that schemas alone are insufficient to provide an overall "big picture" of the data available and how they can be mined for insights. We've frequently observed that data scientists get stuck before they even begin—it's surprisingly difficult in a large production environment to understand what data exist, how they are structured, and how they relate to each other. Our discussion is couched in the context of user behavior logs, which comprise the bulk of our data. We share a number of examples, based on our experience, of what doesn't work and how to fix it.

Second, we observe that a major challenge in building data analytics platforms comes from the heterogeneity of the various components that must be integrated together into production workflows. Much complexity arises from impedance

mismatches at the interface between different components. A production system must run like clockwork, splitting out aggregated reports every hour, updating data products every three hours, generating new classifier models daily, etc. Getting a bunch of heterogeneous components to operate as a synchronized and coordinated workflow is challenging. This is what we fondly call "plumbing"—the not-so-sexy pieces of software (and duct tape and chewing gum) that ensure everything runs together smoothly is part of the "black magic" of converting data to insights.

This paper has two goals: For practitioners, we hope to share our experiences to flatten bumps in the road for those who come after us. Scaling big data infrastructure is a complex endeavor, and we point out potential pitfalls along the way, with possible solutions. For academic researchers, we hope to provide a broader context for data mining in production environments—to help academics understand how their research is adapted and applied to solve real-world problems at scale. In addition, we identify opportunities for future work that could contribute to streamline big data mining infrastructure.

2. HOW WE GOT HERE

We begin by situating the Twitter analytics platform in the broader context of "big data" for commercial enterprises. The "fourth paradigm" of how big data is reshaping the physical and natural sciences [23] (e.g., high-energy physics and bioinformatics) is beyond the scope of this paper.

The simple idea that an organization should retain data that result from carrying out its mission and exploit those data to generate insights that benefit the organization is of course not new. Commonly known as business intelligence, among other monikers, its origins date back several decades. In this sense, the "big data" hype is simply a rebranding of what many organizations have been doing all along.

Examined more closely, however, there are three major trends that distinguish insight-generation activities today from, say, the 1990s. First, we have seen a tremendous explosion in the sheer amount of data—orders of magnitude increase. In the past, enterprises have typically focused on gathering data that are *obviously* valuable, such as business objects representing customers, items in catalogs, purchases, contracts, etc. Today, in addition to such data, organizations also gather behavioral data from users. In the online setting, these include web pages that users visit, links that they click on, etc. The advent of social media and user-generated content, and the resulting interest in encouraging such interactions, further contributes to the amount of data that is being accumulated.

Second, and more recently, we have seen increasing sophistication in the types of analyses that organizations perform on their vast data stores. Traditionally, most of the information needs fall under what is known as online analytical processing (OLAP). Common tasks include ETL (extract, transform, load) from multiple data sources, creating joined views, followed by filtering, aggregation, or cube materialization. Statisticians might use the phrase *descriptive statistics* to describe this type of analysis. These outputs might feed report generators, front-end dashboards, and other visualization tools to support common "roll up" and "drill down" operations on multi-dimensional data. Today, however, a new breed of "data scientists" want to do far more: they are interested in *predictive analytics*. These include, for example, using machine learning techniques to train predictive models of user behavior—whether a piece of content is spam, whether two users should become "friends", the likelihood that a user will complete a purchase or be interested in a related product, etc. Other desired capabilities include mining large (often unstructured) data for statistical regularities, using a wide range of techniques from simple (e.g., k-means clustering) to complex (e.g., latent Dirichlet allocation or other Bayesian approaches). These techniques might surface "latent" facts about the users—such as their interest and expertise—that they do not explicitly express.

To be fair, some types of predictive analytics have a long history—for example, credit card fraud detection and market basket analysis. However, we believe there are several qualitative differences. The application of data mining on behavioral data changes the scale at which algorithms need to operate, and the generally weaker signals present in such data require more sophisticated algorithms to produce insights. Furthermore, expectations have grown—what were once cutting-edge techniques practiced only by a few innovative organizations are now routine, and perhaps even necessary for survival in today's competitive environment. Thus, capabilities that may have previously been considered luxuries are now essential.

Finally, open-source software is playing an increasingly important role in today's ecosystem. A decade ago, no credible open-source, enterprise-grade, distributed data analytics platform capable of handling large data volumes existed. Today, the Hadoop open-source implementation of MapReduce [11] lies at the center of a *de facto* platform for large-scale data analytics, surrounded by complementary systems such as HBase, ZooKeeper, Pig, Hive, and many others. The importance of Hadoop is validated not only by adoption in countless startups, but also the endorsement of industry heavyweights such as IBM, Microsoft, Oracle, and EMC. Of course, Hadoop is not a panacea and is not an adequate solution for many problems, but a strong case can be made for Hadoop supplementing (and in some cases, replacing) existing data management systems.

Analytics at Twitter lies at the intersection of these three developments. The social media aspect is of course obvious. Like Facebook [20], LinkedIn, and many other companies, Twitter has eschewed, to the extent practical, costly proprietary systems in favor of building around the Hadoop open-source platform. Finally, like other organizations, analytics at Twitter range widely in sophistication, from simple aggregations to training machine-learned models.

3. THE BIG DATA MINING CYCLE

In production environments, effective big data mining at scale doesn't begin or end with what academics would consider data mining. Most of the research literature (e.g., KDD papers) focus on better algorithms, statistical models, or machine learning techniques—usually starting with a (relatively) well-defined problem, clear metrics for success, and existing data. The criteria for publication typically involve improvements in some figure of merit (hopefully statistically significant): the new proposed method is more accurate, runs faster, requires less memory, is more robust to noise, etc.

In contrast, the problems we grapple with on a daily basis are far more "messy". Let us illustrate with a realistic but hypothetical scenario. We typically begin with a poorly formulated problem, often driven from outside engineering

and aligned with strategic objectives of the organization, e.g., "we need to accelerate user growth". Data scientists are tasked with executing against the goal—and to operationalize the vague directive into a concrete, solvable problem requires exploratory data analysis. Consider the following sample questions:

- When do users typically log in and out?
- How frequently?
- What features of the product do they use?
- Do different groups of users behave differently?
- Do these activities correlate with engagement?
- What network features correlate with activity?
- How do activity profiles of users change over time?

Before beginning exploratory data analysis, the data scientist needs to know what data are available and how they are organized. This fact may seem obvious, but is surprisingly difficult in practice. To understand why, we must take a slight detour to discuss service architectures.

3.1 Service Architectures and Logging

Web- and internet-based products today are typically designed as composition of services: rendering a web page might require the coordination of dozens or even hundreds of component services that provide different content. For example, Twitter is powered by many loosely-coordinated services, for adding and removing followers, for storing and fetching tweets, for providing user recommendations, for accessing search functionalities, etc. Most services are not aware of the implementation details of other services and all services communicate through well-defined and stable interfaces. Typically, in this design, each service performs its own logging—these are records of requests, along with related information such as who (i.e., the user or another service) made the request, when the request occurred, what the result was, how long it took, any warnings or exceptions that were thrown, etc. The log data are independent of each other by design, the side effect of which is the creation of a large number of isolated data stores which need to be composed to reconstruct a complete picture of what happened during processing of even a single user request.

Since a single user action may involve many services, a data scientist wishing to analyze user behavior must first identify all the disparate data sources involved, understand their contents, and then join potentially incompatible data to reconstruct what users were doing. This is often easier said than done! In our Hadoop data warehouse, logs are all deposited in the `/logs/` directory, with a sub-directory for each log "category". There are dozens of log categories, many of which are named after projects whose function is well-known but whose internal project names are not intuitive. Services are normally developed and operated by different teams, which may adopt different conventions for storing and organizing log data. Frequently, engineers who build the services have little interaction with the data scientists who analyze the data—so there is no guarantee that fields needed for data analysis are actually present (see Section 4.5). Furthermore, services change over time in a number of ways: functionalities evolve; two services merge into one; a single service is replaced by two; a service becomes obsolete and is replaced by another; a service is used in ways for which it was not originally intended. These are all reflected in idiosyncrasies present in individual service logs—these comprise the "institutional" knowledge that data scientists acquire over time, but create steep learning curves for new data scientists.

The net effect is that data scientists expend a large amount of effort to understand the data available to them, before they even begin any meaningful analysis. In the next section, we discuss some solutions, but these challenges are far from solved. There is an increasing trend to structure teams that are better integrated, e.g., involving data scientists in the design of a service to make sure the right data are captured, or even having data scientists embedded within individual product teams, but such organizations are the exception, not the rule. A fundamental tradeoff in organizational solutions is development speed vs. ease of analysis: incorporating analytics considerations when designing and building services slows down development, but failure to do so results in unsustainable downstream complexity. A good balance between these competing concerns is difficult to strike.

3.2 Exploratory Data Analysis

Exploratory data analysis *always* reveals data quality issues. In our recollection, we have never encountered a large, real-world dataset that was directly usable without data cleaning. Sometimes there are outright bugs, e.g., inconsistently formatted messages or values that shouldn't exist. There are inevitably corrupt records—e.g., a partially written message caused by a premature closing of a file handle. Cleaning data often involves sanity checking: a common technique for a service that logs aggregate counts as well as component counts is to make sure the sum of component counts matches the aggregate counts. Another is to compute various frequencies from the raw data to make sure the numbers seem "reasonable". This is surprisingly difficult—identifying values that seem suspiciously high or suspiciously low requires experience, since the aggregate behavior of millions of users is frequently counter-intuitive. We have encountered many instances in which we thought that there must have been data collection errors, and only after careful verification of the data generation and import pipeline were we confident that, indeed, users did really behave in some unexpected manner.

Sanity checking frequently reveals abrupt shifts in the characteristics of the data. For example, in a single day, the prevalence of a particular type of message might decrease or increase by orders of magnitude. This is frequently an artifact of some system change—for example, a new feature just having been rolled out to the public or a service endpoint that has been deprecated in favor of a new system. Twitter has gotten sufficiently complex that it is no longer possible for any single individual to keep track of every product rollout, API change, bug fix release, etc., and thus abrupt changes in datasets appear mysterious until the underlying causes are understood, which is often a time-consuming activity requiring cross-team cooperation.

Even when the logs are "correct", there are usually a host of outliers caused by "non-typical" use cases, most often attributable to non-human actors in a human domain. For example, in a query log, there will be robots responsible for tens of thousands of queries a day, robots who issue queries thousands of terms long, etc. Without discarding these outliers, any subsequent analysis will produce skewed results. Although over time, a data scientist gains experi-

ence in data cleaning and the process becomes more routine, there are frequently surprises and new situations. We have not yet reached the point where data cleaning can be performed automatically.

3.3 Data Mining

Typically, after exploratory data analysis, the data scientist is able to more precisely formulate the problem, cast it in within the context of a data mining task, and define metrics for success. For example, one way to increase active user growth is to increase retention of existing users (in addition to adding new users): it might be useful to build a model that predicts *future* user activity based on present activity. This could be more precisely formulated as a classification problem: assuming we have a definition of an "active user", given features of the user right now, let us try to predict if the user will be active n weeks from now. The metrics of success are now fairly straightforward to define: accuracy, precision–recall curves, etc.

With a precisely-formulated problem in hand, the data scientist can now gather training and test data. In this case, it is fairly obvious what to do: we could use data from n weeks ago to predict if the user is active *today*. Now comes the part that would be familiar to all data mining researchers and practitioners: feature extraction and machine learning. Applying domain knowledge, the data scientist would distill potentially tens of terabytes of log data into much more compact sparse feature vectors, and from those, train a classification model. At Twitter, this would typically be accomplished via Pig scripts [42, 15] that are compiled into physical plans executed as Hadoop jobs. For a more detailed discussion of our large-scale machine learning infrastructure, we refer the reader to a recent paper [34].

The data scientist would now iteratively refine the classifier using standard practices: cross-validation, feature selection, tuning of model parameters, etc. After an appropriate level of effectiveness has been achieved, the classifier might be evaluated in a prospective manner—using data from today and verifying prediction accuracy n weeks from now. This ensures, for example, that we have not inadvertently given the classifier future information.

At this point, let us suppose that we have achieved a high level of classifier effectiveness by some appropriate metric, on both cross-validated retrospective data and on prospective data in a simulated deployment setting. For the academic researcher, the problem can be considered "solved": time to write up the experiments in a KDD paper!

3.4 Production and Other Considerations

However, from the Twitter perspective, there is much left to do: the classifier has not yet been productionized. It is not sufficient to solve the problem once—we must set up recurring workflows that feed new data to the classifier and record its output, serving as input to other downstream processes. This involves mechanisms for scheduling (e.g., running classification jobs every hour) and data dependency management (e.g., making sure that upstream processes have generated necessary data before invoking the classifiers). Of course, the workflow must be robust and continuously monitored, e.g., automatic restarts for handling simple faults, but alerting on-call engineers after a number of failed retries. Twitter has developed tools and processes for these myriad issues, and handling most scenarios are quite routine today, but building the production support infrastructure required substantial engineering effort.

Moving a classifier into production also requires retraining the underlying model on a periodic basis and some mechanism for validation. Over time, we need to manage two challenges: classifier drift and adversarial interactions. User behaviors change, sometimes as a result of the very system we're deploying (e.g., user recommendations alter users' linking behavior). Features that were previously discriminative may decay in their effectiveness. The underlying class distribution (in the case of classification tasks) also changes, thus negating parameters tuned for a specific prior. In addition to classifier drift that stems from "natural" behavioral shifts, we must also contend with adversarial interactions, where third parties actively try to "game" the system—spam is the most prominent example, but we see adversarial behavior elsewhere as well [48]. A data scientist is responsible for making sure that a solution "keeps working".

After a product has launched, data scientists incrementally improve the underlying algorithms based on feedback from user behavior. Improvements range from simple parameter tuning to experimenting with different algorithms. Most production algorithms are actually ensembles that combine diverse techniques. At Twitter, as in many organizations today, refinements are assessed via A/B testing. How to properly run such experiments is as much an art as it is a science, and for the interested reader we recommend a few papers by Kohavi et al. [29, 28]. From the big data infrastructure perspective, this places additional demands on tools to support A/B testing—e.g., identifying user buckets and keeping track of user-to-treatment mappings, as well as "threading" the user token through all analytics processes so that we can break down results by each condition.

Finally, the successful deployment of a machine-learned solution or any data product leads to the start of a new problem. In our running example of retention classification, being able to predict user activity itself doesn't actually affect user growth (the original goal)—we must act on the classifier output to implement interventions, and then measure the effectiveness of those. Thus, one big data mining problem feeds into the next, beginning the cycle anew.

In this production context, we identify two distinct but complementary roles: on the one hand, there are infrastructure engineers who build the tools and focus on operations; then, there are the data scientists who use the tools to mine insights. Although in a smaller organization the same person may perform both types of activities, we recognize them as distinct roles. As an organization grows, it makes sense to separate out these two activities. At Twitter, analytics infrastructure and data science are two distinct, but tightly-integrated groups.

To better illustrate this division, consider the retention classifier example. The data scientist would be responsible for the development of the model, including exploratory data analysis, feature generation, model construction, and validation. However, she would be using tools built by the infrastructure engineers (e.g., load and store functions for various data types), operating on data maintained by infrastructure engineers (responsible for the log import pipeline). The data scientist is responsible for the initial production deployment of the model (e.g., hooking into a common set of APIs to specify data dependencies and setting up recurring workflows) but thereafter the infrastructure engineers

handle routine operations—they are responsible for "keeping the trains on time". However, data scientists are responsible for maintaining the quality of their products after launch, such as handling classifier drift and adversarial interactions. Overall, we have found this division between infrastructure engineering and data science to be a useful organizing principle in scaling our data analytics platform.

3.5 Why Should Academics Care?

The takeaway message from this discussion is that activities an academic researcher would consider data mining (e.g., feature extraction and machine learning), while important, are only a small part of the complete data mining cycle. There is much that precedes in formulating the problem, data cleaning, and exploratory data analysis; there is much that follows, in terms of productionizing the solution and ongoing maintenance. Any big data mining infrastructure must support all activities in this broader lifecycle, not just the execution of data mining algorithms.

Why should academics care? We argue that understanding the big data mining cycle helps situate research in the context of solving real-world problems and informs future directions. For example, work on data cleaning and exploratory data analysis is under-represented in the academic literature relative to its real-world importance. We suggest that the most impactful research is not incrementally better data mining algorithms, but techniques to help data scientists "grok" the data. Furthermore, production considerations might help researchers select between competing methods. A lesson from the above discussion might be that complex models with many parameters are difficult to maintain, due to drift and adversarial interactions. Another might be that the need to frequently update models favor those that can be incrementally refined, as opposed to those that must be trained from scratch each time. Different models and algorithms manifest tradeoffs in accuracy, robustness, complexity, speed, etc., and understanding the deployment context is important to choosing the right balance point in the design space.

We conclude this section by relating our experience to those of others. While the literature on these practical, "in-the-trenches" aspects of data science is scant, we are heartened to see a growing awareness of these important issues. One of the earliest is Jeff Hammerbacher's essay on the construction of Facebook's data science platform [20]. DJ Patil has written extensively on his experiences in building data-driven products at LinkedIn [46, 45]. Recently, Kandel et al. [27] reported on an interview-based study of data scientists across a variety of sectors, including healthcare, retail, marketing, and finance. Their findings are consonant with our own experiences, although specific details differ. Together, we hope that these voices provide a more complete account of what big data mining is like in real-world production environments.

4. SCHEMAS AND BEYOND

Typically, what we've seen in the past is that an enterprise's most obviously valuable data, such as business objects representing customers, items in catalogs, purchases, contracts, etc. are represented in carefully designed schemas. The quality, usefulness, and provenance of these records are carefully monitored. However, higher-volume data such as operational logs of the multitude of services that power a complex operation are often ad hoc. A well-structured log should capture information about each request, such as who made the request, when the request occurred, what the result was, how long it took, any warnings or exceptions that were thrown. Frequently, however, they are merely reflections of what individual developers decided would be helpful to throw into their `log.info(...)` calls. Since these logs contain records of user behavior, they provide valuable raw ingredients for data mining. From the perspective of infrastructure engineering, we are faced with the challenge of supporting rapid development of individual services, with little to no oversight from a central authority, while enabling analysis of diverse log sources by data scientists across the organization.

```
create table `my_audit_log` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `created_at` datetime,
  `user_id` int(11),
  `action` varchar(256),
  ...
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Table 1: Using MySQL for logging is a bad idea.

Some organizations attempt to solve this problem by enforcing simple, generic schemas for all the data that the developers "explicitly want to track"[1] which, from what we can tell, amounts to little more than key–value or subject–verb–object abstractions (see Section 4.3). We believe that insisting on such an approach reduces the expressivity of queries and limits the organization's ability to answer unanticipated questions.

Our advice distills experiences in balancing concerns about introducing friction into the development process and the desire to enable easy access to a great variety of data sources across the enterprise. We seek to maximize the opportunities for insightful data analysis, data mining, and the construction of data products, while minimizing the amount of people in the critical communication paths.

Below, we critique some common techniques for capturing data for subsequent analysis and data mining. We do so mostly from a scaling perspective, with full understanding that for smaller organizations, some argument may not be applicable. A small team can be quite successful using JSON; a small rails shop that doesn't see much traffic might be fine directly logging into the database. Note that by no means will our suggestions solve all the challenges discussed in the previous section, but we believe that our recommendations will address some of the common pain points in building scalable big data infrastructure.

4.1 Don't Use MySQL as a Log

Logging directly into a database is a very common pattern, arguably a natural one when the main application already has a dependency on an RDBMS. It is a good design if the database can keep up: no new systems to worry about, a plethora of tools to query the data, and the logs are always up to date. The design is tantalizingly simple: just create a flexible schema such as, for example, that shown in Table 1. Using a database for logging at scale, however, quickly breaks down for a whole host of reasons.

[1] http://code.zynga.com/2011/06/deciding-how-to-store-billions-of-rows-per-day/

First, there is a mismatch between logging and typical database workloads for interactive or user-facing services: the latter are almost always latency sensitive and usually involve relatively small individual transactions. In contrast, logging creates heavy write traffic. Co-locating the two different workloads on the same system usually results in poor performance for both, even if the databases are fronted by in-memory caches (as is the standard architecture). For the same reason, the convenience of storing logs in the database for log analysis is not meaningful in practice, since complex long-running analytical queries (that may scan through large amounts of data) conflict with latency-sensitive user queries, unless they are isolated to dedicated resources, for example, via setting up "analytics-only" replicas.

Second, scaling databases is a non-trivial challenge. In the open-source world, one can easily find on the web a plethora of blog posts and war stories about the pains of running sharded MySQL at scale. Having to scale *both* a user-facing service and the logging infrastructure makes the problem unnecessarily complex. Organizations working with proprietary databases face a different set of challenges: proprietary offerings can be more scalable, but are costly and often require hiring highly-specialized administrators.

Third, databases are simply overkill for logging. They are built for transactions and mutability of data, both of which come with performance overhead. Neither is necessary for logging, so there is nothing to be gained from paying this performance penalty—in fact, the ability to mutate log records after they are written invites an entirely new class of bugs. Logs should be treated as immutable, append-only data sinks.

Fourth, evolving schemas in relational databases is often a painful and disruptive process. For example, most `ALTER` operations in MySQL cause the table to be locked and rebuilt, which can be a long process—this essentially means down time for the service. Schemas for logs naturally evolve, since frequently one realizes the need to gather additional data that was not previously anticipated. As mentioned earlier, one common approach around this problem is to define the table schema around arbitrary key-value pairs, which to a large extent defeats the point of having a schema and using a database in the first place (see discussion below on JSON).

The database community has long realized most of these points, which is why database architectures have evolved separate systems for OLTP (online transaction processing) and OLAP (online analytical processing). A discussion of traditional OLAP systems is beyond the scope of this paper. Note that when we advocate against using the database as a log, we certainly do not advocate against traditional log ETL processes feeding into OLAP databases—just against writing log messages directly to a database from the application layer.

4.2 The Log Transport Problem

If databases shouldn't be used for logging, what then? At a basic level, we need a mechanism to move logs from where they are generated—machines running service endpoints—to where they will ultimately be analyzed, which for Twitter as well as many organizations is the Hadoop Distributed File System (HDFS) of a Hadoop cluster.

We consider the problem of log transport mostly solved. Twitter uses Scribe, a system for aggregating high volumes of streaming log data in a robust, fault-tolerant, distributed

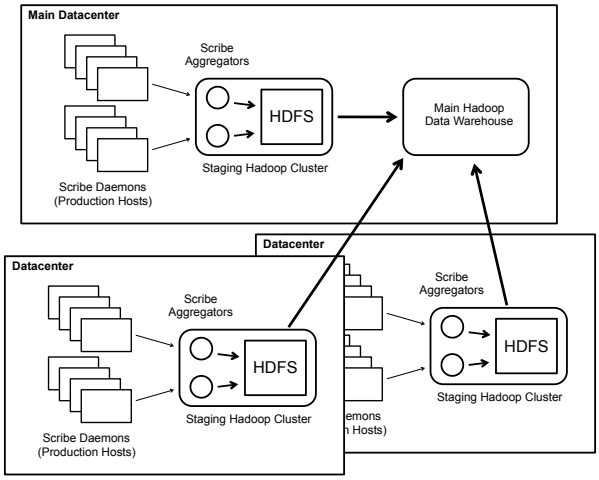

Figure 1: Illustration of Twitter's Scribe infrastructure. Scribe daemons on production hosts send log messages to Scribe aggregators, which deposit aggregated log data onto per-datacenter staging Hadoop clusters. Periodic processes then copy data from these staging clusters into our main Hadoop data warehouse.

manner, originally developed by Facebook and now an open-source project. A description of how Scribe fits into Facebook's overall data collection pipeline can be found in [51].

Twitter's Scribe infrastructure is illustrated in Figure 1; see [31] for more details. A Scribe daemon runs on every production host and is responsible for sending local log data across the network to a cluster of dedicated aggregators in the same datacenter. Each log entry consists of two strings, a category and a message. The category is associated with configuration metadata that determine, among other things, where the data is written.

The aggregators in each datacenter are co-located with a staging Hadoop cluster. Their task is to merge per-category streams from all the server daemons and write the merged results to HDFS (of the staging Hadoop cluster), compressing data on the fly. Another process is responsible for moving these logs from the per-datacenter staging clusters into the main Hadoop data warehouse. It applies certain sanity checks and transformations, such as merging many small files into a few big ones and building any necessary indexes.

A few additional issues to consider: a robust log transfer system may create large amounts of network traffic, thus the use of CPU-intensive log compression techniques should be balanced against the availability of network bandwidth. As systems scale up, a robust log transport architecture needs to build in tolerances for rack switch failures and other unexpected network glitches—at some point, the architecture needs of a massive log collection system become non-trivial.

Within Twitter, at the end of the log mover pipeline, data arrive in the main Hadoop data warehouse and are deposited in per-category, per-hour directories on HDFS (e.g., `/logs/category/YYYY/MM/DD/HH/`). In each directory, log messages are bundled in a small number of large files.

The bottom line: log transport at scale is a well-understood problem, and Scribe is a mature, scalable solution. More recent open-source systems such as Apache Flume[2] and Apache

[2] http://flume.apache.org/

```
^(\\w+\\s+\\d+\\s+\\d+:\\d+:\\d+)\\s+
([^@]+?)@(\\S+)\\s+(\\S+):\\s+(\\S+)\\s+(\\S+)
\\s+((?:\\S+?,\\s+)*(?:\\S+?))\\s+(\\S+)\\s+(\\S+)
\\s+\\[([^\\]]+)\\]\\s+\"(\\w+)\\s+([^\"\\\\]*
(?:\\\\.[^\"\\\\]*)*)\\s+(\\S+)\"\\s+(\\S+)\\s+
(\\S+)\\s+\"([^\"\\\\]*(?:\\\\.[^\"\\\\]*)*)
\"\\s+\"([^\"\\\\]*(?:\\\\.[^\"\\\\]*)*)\"\\s*
(\\d*-[\\d-]*)?\\s*(\\d+)?\\s*(\\d*\\.[\\d\\.]*)?
(\\s+[-\\w]+)?.*$
```

Table 2: An actual Java regular expression used to parse log messages at Twitter circa 2010.

Kafka [30] offer similar functionality—we consider using one of these systems to be best practice today.[3]

4.3 From Plain Text to JSON

Having addressed log transport, let us move onto the next question: How should log messages be structured?

The instinctive starting point for many developers is to capture the standard logs their application generates, be it as simple as the `stdout` and `stderr` streams, or information collected via dedicated logging libraries such as Java's Log4J. Either way, this means plain-text log messages, the most common case being delimited fields with a record per line, although multi-line messages such as stack traces are frequently seen as well. A parsing and maintenance nightmare develops quickly. Pop quiz: what's the delimiter, and would everyone in the organization give the same answer?

Delimited, line-oriented plain-text records require careful escaping of special characters, which is riddled with unexpected corner cases. There is a memorable example of us having discovered that some Twitter user names contain embedded newlines and tabs, which needless to say was not a condition we checked for, and therefore led to corrupt records in a processing script.

The next pain point: parsing plain-text log messages is both slow and error-prone. Regular expression matching, splitting strings based on delimiters, and converting strings back to integers or floating point values are all slow operations, which means that working with plain-text logs is inefficient. Furthermore, regular expressions for parsing complex log messages are difficult to understand, maintain, and extend. An extreme, but perhaps not atypical, example is the Java regular expression shown in Table 2, which was used for some time in the Twitter code base (circa 2010)—made extra fun due to double-escaping slashes required for Java strings. Mercifully, this code has been retired.

Having understood the downsides of working with plain-text log messages, the next logical evolution step is to use a loosely-structured, easy-to-evolve markup scheme. JSON is a popular format we've seen adopted by multiple companies, and tried ourselves. It has the advantage of being easily parseable, arbitrarily extensible, and amenable to compression. JSON log messages generally start out as an adequate solution, but then gradually spiral into a persistent nightmare. We illustrate with a hypothetical but realistic JSON log message in Table 3. Let us run through a thought exercise in which we discuss the depressing but all-too-realistic

[3]Since Kafka is a distributed queuing system, adopting it for logging requires a different architecture, see [17]. The general design is for production hosts to write log messages to Kafka, and then implement a periodic process (e.g., every ten minutes) to "roll up" log messages and persist to HDFS.

```
{
  "token": 945842,
  "feature_enabled": "super_special",
  "userid": 229922,
  "page": "null",
  "info": { "email": "my@place.com" }
}
```

Table 3: A hypothetical JSON log message.

ways in which this example might be broken.

Trouble begins with an issue as simple as naming conventions. Building a successful web service today requires bringing together developers with very diverse skill sets: front-end, back-end, systems, databases, etc. Developers accustomed to different language environments have conflicting conventions for naming fields and variables. This results in code for generating and analyzing logs being littered with `CamelCase`, `smallCamelCase`, as well as `snake_case`, and occasionally, even the dreaded `camel_Snake`.[4] To make matters worse, we've even encountered at least one instance of the mythical `dunder__snake`[5] in the Twitter code base.[6] As a specific example: Is the user id field `uid`, `userId`, `userid`, or `user_id` (of course, not ruling out `user_Id`)? In the case of JSON, it usually suffices to examine a few records to figure out the field names, but this is one more annoyance that gets in the way of productive analysis. Sometimes, however, bugs with field names crop up in subtle ways: for example, different code paths might accidentally adopt different conventions, resulting in some of the log messages having one field, and some of the log messages the other (or, even worse, log messages containing *both* fields).

Consider the property `feature_enabled`. It should really have been a list, but we didn't anticipate users being opted into more than one feature. Now that someone pointed this out (perhaps a developer who is adding a new feature), the bug will be fixed (tomorrow). In the meantime we'll store multiple values as a comma-separated list. Of course, the comma delimiter isn't documented anywhere, so the data scientist inadvertently uses semi-colon as the delimiter, and naturally, none of the values match expected ones. Meanwhile, tomorrow never comes, and so this hack persists in the code base for years. Or, worse yet, tomorrow comes—and now the data scientist has to discover, thorough some bug database or wiki page "data mining" perhaps, that `feature_enabled` can be a comma-separated list, a semi-colon separated list, *or* an actual JSON array—and now she has to write code to handle all three cases.

The property `userid` (note, not `user_id` or `userId`) is actually supposed to be a string, but this user's id happens to be a sequence of digits, so some upstream process casted it into a JSON number. Now the type is inconsistent across different instances of the same message type. Any downstream process that attempts to perform a join will throw a type mismatch exception. Next issue: consider the value of the `page` property—is it really the string "null"? Or is it actually supposed to be `null`? Or `nil`? This may seem pedantic but is nevertheless important. At a recent conference, when presenting this point, one of the authors asked the developers in the audience to raise their hand if their

[4]https://twitter.com/billgraham/status/245687780640960514
[5]http://wiki.python.org/moin/DunderAlias
[6]https://twitter.com/billgraham/status/246077392408412160

time has been wasted in the past chasing down bugs that turned out to be `null`s represented as strings when parsing JSON. Almost everyone raised their hand. We don't have to live like this. There is a better way.

Finally, the value of the `info` property is a nested object. Is it a flat key–value map, or can there be deeper nesting? What fields are obligatory, what fields are optional? For each field, what is the type and range of values? Can it be null, and if so, how is it indicated (see issue above)? In many cases, even obtaining a complete catalog of all possible fields is difficult. Internal documentation is almost always out of date, and the knowledge lives primarily in the team of developers who created the applications (who themselves are often fuzzy on the details of code they had written months ago). To get around this issue, data scientists often have to read code (sometimes, old versions of the code dug up from source control!) to figure out the peculiarities of a log message format. Another common "solution" is to induce the message format manually by writing Pig jobs that scrape large numbers of messages to produce key–value histograms. Needless to say, both of these alternatives are slow and error-prone. Note that the same arguments can be applied to the top-level JSON structure, but nested objects make all these issues even more maddening.

A large part of this particular issue stems from JSON's lack of a fixed schema. This affords the developer great flexibility in describing complex data with arbitrarily nested structures, but causes headaches for data scientists who must make sense of log semantics downstream. Certainly, one can posit a method for sharing JSON schemas—and we have gone down this path in the past, with XML and DTDs–but at that point, why bother with JSON?

4.4 Structured Representations

As described in the previous section, data representation formats such as JSON standardize parsing but still allow too much (unconstrained!) flexibility, making long-term data analysis difficult. We would like to make data scientists' lives easier without burdening application developers upstream, for example, by forcing them into overly-rigid structures. Like many design problems, striking a good balance between expressivity and simplicity is difficult.

Our approach to address these problems has been to institutionalize the use of Apache Thrift for serialization of all logged messages. Thrift[7] provides a simple grammar for declaring typed structs and code generation tools to allow the creation and consumption of such structs in a number of programming languages. Binary serialization protocols avoid the performance penalty of repeatedly parsing values from string-based representations. Another key feature of Thrift is the ability to specify optional and required fields, which provides a sane path for schema evolution (more below). We have developed a set of tools collectively called Elephant Bird[8] that hooks into the Thrift serialization compiler to automatically generate record readers, record writers, and other "glue" code for both Hadoop, Pig, and other tools. Protocol Buffers [12] (open-sourced by Google[9]) and Apache Avro[10] achieve similar goals, with minor implementation differences.

[7] http://thrift.apache.org/
[8] http://github.com/kevinweil/elephant-bird
[9] http://code.google.com/p/protobuf/
[10] http://avro.apache.org/

```
struct MessageInfo {
  1: optional string name
  2: optional string email  // NOTE: unverified.
}

enum Feature {
  super_special,
  less_special
}

struct LogMessage {
  1: required i64 token
  2: required string user_id
  3: optional list<Feature> enabled_features
  4: optional i64 page = 0
  5: optional MessageInfo info
}
```

Table 4: Message definition in Apache Thrift.

A Thrift type definition of the JSON message shown earlier can be found in Table 4. Here, we clearly and concisely describe constraints on possible messages such as field types (thus reducing type errors), specify which fields are optional and which are required, and explicitly define enums that constrain the range of possible values (when possible).

Combining universal use of Thrift for logging with a small investment in tooling reduces the friction involved in data analysis. With a bit of metadata, all our tools now know how to access any log source. A data scientist can consult the Thrift definition and immediately knows what to expect, with some degree of assurance that the messages will pass basic sanity checks. Schemas can be evolved by adding additional optional fields and deprecating old fields; history of such changes is available via the source control system, and constrained to just a couple of schema definition files. Note that although Thrift does not completely resolve the ongoing struggles between the camel and the snake (i.e., different naming conventions), at least now there is never any doubt as to what the fields are *actually* named.

Another benefit of using something like Thrift or Avro is that it provides a separation between the logical and physical data representations, a property long appreciated in the database community. This separation allows the storage and infrastructure teams to change physical representations of the data on disk, experiment with different compression algorithms to tune speed/capacity tradeoffs, put data into columnar stores [40, 21, 36, 24], create indexes [13, 35], and apply a number of other optimizations. As long as the messages can be reconstructed in accordance with the declared schema, consumers and producers do not need to be concerned with what happens "under the hood". While the simplicity of dumping JSON to flat files can allow a small team to iterate quickly, the infrastructure we have described provides many benefits when working with large numbers of data sources, diverse data consumers of varying technical sophistication, and sufficiently large volumes that advanced data storage and compression techniques become important.

4.5 Looking Beyond the Schema

While access to schemas reduce much unnecessary overhead when working with diverse datasets, schemas by themselves are not, of course, sufficient for frictionless data analysis at scale. A complete *data catalog* is needed, which would

allow data scientists, product managers, and other interested parties to explore what is available in the data warehouse without blindly poking around Git repositories filled with Thrift IDL files. The basis of such a service can be found in the HCatalog project in the Apache Incubator. HCatalog provides a basic "table" abstraction for files stored in HDFS, and, true to its name, maintains a catalog of all registered tables and their schemas. Tables may contain nested objects, and there is good support for Thrift and Avro data sources. As long as all processes that create or alter data on the Hadoop cluster do so through the HCatalog APIs, it is possible to create web-based tools for exploring the complete set of available data resources. Having such a centralized catalog is invaluable, but integration has proven to be surprisingly difficult when our team tried to bolt it onto existing workflows post-factum, since hundreds of scripts had to be converted to use HCatalog, many of them manifesting corner cases that required special treatment. Here lies a dilemma: for a small team working with only a few data sources, the overhead of using HCatalog may be more trouble than it's worth. However, as the complexity of the analytics platform grows, so does the need for the capabilities that HCatalog offers—but adapting existing workflows can be painful. There is no simple resolution, as the "right" approach is heavily context-dependent.

At Twitter, we found that provenance data, not currently covered by HCatalog, are extremely useful for answering both data mining questions ("What source data was used to produce these user clusters?") as well as questions related to troubleshooting and recovery ("What consumed this incorrect data? Should we regenerate downstream products?"). In order to add provenance tracking to HCatalog, we built a thin wrapper around HCatalog loaders and storers that records information about all reads and writes to different data sets. These access traces can then be used to construct a complete—and correct—graph of job dependencies based on the *actual* data access patterns. Other approaches to solving the dependency graph visibility problem require users to explicitly construct such graphs while structuring their workflows. We found this to be onerous for the user and error-prone, as some reads and writes invariably happen "out-of-band" and are not explicitly declared. When the graph is constructed directly from the executing workflow, the possibility of missing edges is reduced. In the future, we plan on further integrating this metadata layer with our source control and deployment systems so we can automatically identify likely owners of different applications and to trace problems to individual code commits or deploy versions when troubleshooting production issues.

Having posited the availability of such a metadata service, let us now consider a hypothetical scenario in which a data scientist wishes to understand how a particular product is used. Because of the service architecture discussed in Section 3.1, the necessary data is almost always distributed across multiple log types. With HCatalog and all of the best practices discussed above, it should be easy for her to identify the right data sources and gain an overview of their contents quickly. However, the challenge occurs when she attempts to reconstruct user sessions, which form the basic unit of most analyses. Session reconstruction is typically accomplished by joining log messages from different services, sorting by time, and then segmenting based on an inactivity threshold. But what's the join key? Most naturally, a composite key comprising the user id and a session token (based on a browser cookie) would suffice, provided that the application developer logged the user id and session token! Unfortunately, that information isn't always available. In many cases, it's not even an issue of logging: the service API simply wasn't designed to require the session token in its request invocation, so there's no way to record the information without changing the API.[11] We can usually get around missing data by performing timestamp arithmetic to reconstruct the user's sequences of actions, but such hacks are fraught with complex, unmaintainable temporal logic and full of corner cases that aren't accounted for. On the issue of timestamps, we must frequently deal with conversion between different formats: one source logs epoch milliseconds, the second a timestamp in MySQL format, the third a timestamp in Apache log format. This is a relatively minor annoyance as we have a library for time format conversions, but the accumulation of annoyances increases development friction and slows iteration speed. It would be desirable to have *shared semantics* across different log types, but this is very hard to accomplish in a large organization with dozens of teams working on a multitude of services, largely independent of each other. The upshot is that data scientists spend an inordinate amount of time reconstructing the user's activity before any useful analysis even begins.

"Client events" is an attempt to address this issue for logs generated by Twitter clients (e.g., the twitter.com site, clients on iPhones and Android devices, etc.) by unifying log formats. This project was described in a recent paper [31], so here we only provide a brief summary. The idea behind client event logs is to establish a flexible, shared Thrift schema for all Twitter-owned clients from which data scientists can easily reconstruct user sessions to perform behavior analysis. This is accomplished by requiring that *all* messages share certain fields (e.g., event name, user id, session id, timestamp) with precisely-defined semantics, while allowing sufficient flexibility to capture event-specific details. Event names are modeled after the client's view hierarchy (e.g., DOM in the case of the web client), making it easy to focus on specific Twitter products, e.g., search, discovery, user recommendations, etc. Since the most recent iteration of Twitter clients establishes a common design language, it becomes easier to adapt analysis scripts to, say, compare mobile vs. web usage of the same product. So far, client events have worked well for us and have greatly simplified session-based analyses for data scientists. The takeaway message here is that enforcing some degree of semantic homogeneity can have big payoffs, although we are also quick to point out that this is not possible or practical in many cases.

5. PLUMBING IS IMPORTANT

One of the biggest challenges in building and operating a production analytics platform is handling impedance mismatches that arise from crossing boundaries between different systems and frameworks. Different systems or frameworks excel at different problems, and it makes sense to choose the right tool for the job. However, this must be balanced against the cost of knitting together a patchwork

[11] This happens when a data scientist isn't involved in the design of a service API. For stateless back-end services, there normally wouldn't be a need to keep track of session tokens, except to support downstream analytics.

of different components into integrated workflows. Different systems and frameworks provide alternative models and constructs for thinking about computation: MapReduce forces the developer to decompose everything into *maps* and *reduces*; Pregel [37] makes the developer think in terms of vertex computations and message passing; relational databases are constrained by schemas and SQL. This is not necessarily a bad thing: for example, the simplicity of MapReduce makes it easy to scale out and handle fault tolerance though retries. Different models of computation necessarily make some problems easier (otherwise, there would be no point), but as a side effect they make other problems unnatural or difficult, and in some cases, impossible. Furthermore, systems and frameworks have differing performance and scalability characteristics. Working solely within the confines of a single one, these restrictions are often not very apparent. However, impedance mismatches come into stark relief when trying to cross system or framework boundaries.

Our favorite example of an impedance mismatch is the import pipeline that loads data from user-facing databases into our Hadoop data warehouse. Like many organizations, Twitter runs sharded MySQL databases for most user-facing services. This suggests a very simple import mechanism: connect to the right database partition, issue a `SELECT * FROM...` query, gather the results, and write into HDFS. Of course, we can do this in parallel inside mappers, which speeds up the import process. The problem is that it is very easy to start thousands of mappers in Hadoop—if not careful, we launch what amounts to a denial of service attack against our own system as thousands of processes attempt to simultaneously grab database connections, bringing the database cluster down. The underlying cause is the differing scalability characteristics of MySQL and Hadoop, which are not exposed at the interfaces provided by these services; additional layers for handling QOS, pushing back on over-eager clients (and handling such pushback), etc. become necessary.

The other significant challenge in integrating heterogeneous systems and frameworks for large-scale data processing is threading dataflows across multiple interfaces. The simple fact is that no single production job executes in isolation: everything is part of some larger workflow. The job depends on some data that is generated upstream: the dependencies may be dozens of steps deep and ultimately terminate at data imports from external sources. Similarly, an analytics job usually feeds downstream process, again, potentially dozens of steps deep, ultimately culminating in data that is presented in a dashboard, deployed back out to user-facing services, etc. In a production analytics platform, everything must run like clockwork: data imports must happen at fixed intervals; internal users are expecting dashboards and other reporting mechanisms to be up to date; user-facing data products must be refreshed frequently or the system risks presenting stale data. Orchestrating this entire process, which includes scheduling, dependency management, error handling, monitoring, etc. is no easy task—and is made exponentially more complex if synchronization needs to occur *across* different systems and frameworks.

To further complicate matters, a production analytics platform usually has code contributions from multiple teams. At Twitter, the analytics infrastructure team builds common tools, but most of the analytics workload comes from the dozens of teams who use the tools. The threading of workflows across different systems and frameworks frequently crosses team boundaries—for example, Scalding[12] jobs from the promoted products team depend on the result of graph analytics jobs in Pig built by another team. The analytics infrastructure engineers are responsible for orchestrating this complex operation, but their job is made challenging by the fact that they do not have detailed knowledge of most of the jobs running on Hadoop.

We use the term "plumbing" to refer to the set of challenges presented above. These issues particularly affect data mining at scale, as we explain below.

5.1 Ad Hoc Data Mining

There are three main components of a data mining solution: the raw data, the feature representation extracted from the data, and the model or algorithm used to solve the problem. Accumulated experience over the last decade has shown that in real-world settings, the size of the dataset is the most important factor of the three [18, 33]. Studies have repeatedly shown that simple models trained over enormous quantities of data outperform more sophisticated models trained on less data [2, 7, 14]. For many applications, simple features are sufficient to yield good results—examples in the text domain include the use of simple byte-level n-gram features, eschewing even relatively simple text processing tasks such as HTML tag cleanup and tokenization. This, for example, works well for spam detection [26, 10, 50]. Simple features are fast to extract, thus more scalable: imagine, for example, the computational resources necessary to analyze every single outgoing and incoming message in a web email service. Today, at least in industry, solutions to a wide range of problems are dominated by simple approaches fed with large amounts of data.

For unsupervised data mining, the "throw more data at the problem" approach seems obvious since it allows an organization to make sense of large accumulated data stores. The connection to supervised approaches, however, seems less obvious: isn't the amount of data that can be brought to bear limited by the scale at which ground truth can be acquired? This is where user behavior logs are most useful—the insight is to let users provide the ground truth *implicitly* as part of their normal activities. Perhaps the most successful example of this is "learning to rank" [32], which describes a large class of supervised machine learning approaches to web search ranking. From users' relevance judgments, it is possible to automatically induce ranking functions that optimize a particular loss function. Where do these relevance judgments come from? Commercial search engine companies hire editorial staff to provide these judgments manually with respect to user queries, but they are supplemented with training data mined from search query and interaction logs. It has been shown that certain interaction patterns provide useful (weak) relevance judgments that can be exploited by learning-to-rank techniques [25]. Intuitively, if the user issues a query and clicks on a search result, this observation provides weak evidence that the result is relevant to the query. Of course, user interaction patterns are far more complex and interpretations need to be more nuanced, but the underlying intuition of mining behavior logs to provide (noisy) ground truth for supervised algorithms is foundational and generalizes beyond web search. This creates a virtuous cycle: a high quality product attracts users

[12] https://dev.twitter.com/blog/scalding

and generates a wealth of behavioral data, which can then be mined to further improve the product.

Until relatively recently, the academic data mining and machine learning communities have generally assumed sequential algorithms on data that fit into memory. This assumption is reflected in popular open-source tools such as Weka,[13] Mallet,[14] and others. These tools lower the barrier to entry for data mining, and are widely used by many organizations and other researchers as well. From a pragmatic point of view, it makes little sense to reinvent the wheel, especially for common algorithms such as k-means clustering and logistic regression. However, using tools designed to run on a single server for big data mining is problematic.

To elaborate, we share our experiences working with existing open-source machine learning tools. The broader context is the lifecycle described in Section 3, but here we focus on the actual machine learning. After initial explorations and problem formulation, generation of training and test data for a specific application is performed on Hadoop (typically using Pig). Scripts often process tens of terabytes of data (if not more) to distill compact feature representations; although these are usually orders of magnitude smaller, they can still easily be in the tens to hundreds of gigabytes. Here we encounter an impedance mismatch between a petabyte-scale analytics platform and machine learning tools designed to run on a single machine. The obvious solution is to sample. One simple sampling approach is to generate data from a limited set of logs (e.g., a single day), even though it would be just as easy to generate data from many more. The smaller dataset would then be copied out of HDFS onto another machine (e.g., another server in the datacenter or the developer's laptop). Once the data have been brought over, the developer would perform typical machine learning tasks: training, testing, parameter tuning, etc.

There are many issues with this process. First, and perhaps the most important, is that sampling largely defeats the point of working with big data in the first place. Training a model on only a small fraction of logs does not give an accurate indication of the model's effectiveness at scale. Generally, we observe improvements with growing amounts of data, but there is no way to quantify this short of actually running experiments at scale. Furthermore, sampling often yields biased models—for example, using the most recent day of logs over-samples active users, compared to, for example, analyses across a larger time window. Often, the biases introduced in the sampling process are subtle and difficult to detect. We can remember several instances where a model performed very well on a small sample of users, but its performance steadily deteriorated as we applied it to more and more users.

Second, having to first run Pig scripts and then copy data out of HDFS and import into another environment creates a slow development cycle. Machine learning is an iterative process: for example, after initial experiments we think of a new feature or realize that we should have preprocessed the data in a certain way. This forces a context switch back to Pig to modify the scripts, rerun on the cluster, followed by more data copying. After a few iterations (waiting for jobs on the cluster to complete each time), we are left with several variations of the same dataset in multiple places, and have lost track of which script generated which version. To be fair, we would encounter these data management challenges in any analytics environment, but working across the cluster and a local tool exacerbates the problem. We frequently end up with at least three copies of similar data: the complete dataset on HDFS, one or more smaller datasets on HDFS (sampled using different strategies), and the identical data on a local machine for experimentation.

Finally, productionizing machine-learned models was awkward and brittle at best. Test data were prepared in Pig and divided into small batches, copied out of HDFS, and fed to the learned model (for example, running on another machine in the datacenter). Results were then copied from local disk back to HDFS, where they fed downstream processes. It was most common to have these complex flows orchestrated by shell scripts on cron, which was not a scalable solution. Retraining the model was even more *ad hoc*—in a few cases, we relied on "human cron", or having engineers *remember* to retrain models (by hand) "once in a while".

5.2 Scalable Machine Learning

Over the past few years, scaling up machine learning algorithms with multi-core [41] and cluster-based solutions [1] has received much interest. Examples include learning decision trees and their ensembles [3, 49, 43], MaxEnt models [38], structured perceptrons [39], support vector machines [8], and simple phrase-based approaches [4]. Recent work in online learning (e.g., work by Bottou [6] and Vowpal Wabbit[15]) relax the assumption that data must fit into memory, and are amenable to learning at a massive scale on disk-resident data. In the open-source world, Mahout[16] is emerging as a popular toolkit for large-scale machine learning and data mining tasks.

These are encouraging developments, but not complete end-to-end solutions. In particular, most work focuses on scalable machine learning itself; relatively little attention is paid to the integration issues we have discussed above. For example, Vowpal Wabbit is an extremely fast online learner optimized for streaming through millions of training instances on disk. However, we are still left with the problem of how to get data to the learner—for us, feature generation and data preparation must still be performed on Hadoop, so we still need to orchestrate the process of getting the data out of HDFS and into Vowpal Wabbit. Given the myriad issues discussed above, we hope that the reader realizes by now that this is not an easy task.

Our experience with Mahout raises a different set of issues: there are components of Mahout that are designed to run efficiently on a single machine, and other parts that scale to massive datasets using Hadoop. However, Mahout processing for many tasks consists of monolithic multi-stage pipelines: it demands that data be processed into a specific format and generates results in custom representations. Integrating Mahout into our analytics platform required developing adaptors on both ends—getting data into Mahout and results out. While this was "simply a matter of software engineering" (and indeed we have written and shared much of this "glue" code[17]), it exemplifies the impedance mismatch and integration issues we have been discussing.

[13] http://www.cs.waikato.ac.nz/ml/weka/
[14] http://mallet.cs.umass.edu/
[15] http://hunch.net/∼vw/
[16] http://mahout.apache.org/
[17] https://github.com/kevinweil/elephant-bird

5.3 Integrated Solutions

How do we build big data mining infrastructure that addresses the "plumbing" issues we've discussed above? Here, we share our experiences and discuss alternative approaches.

Our present solution for machine learning was recently presented elsewhere [34], but here we provide a very brief overview. Our approach involves integrating machine learning components into Pig so that machine learning is *just another Pig script*, which allows seamless integration with existing infrastructure for data management, scheduling, and monitoring, as well as access to rich libraries of existing UDFs and the materialized output of other scripts.

This is accomplished with two techniques: stochastic gradient descent for fast one-pass learning and scale out by data partitioning with ensembles. We have developed Pig extensions to fold learners into storage functions, which are abstractions for data sinks that materialize tuples onto durable storage (e.g., HDFS). In our case, instead of materializing the tuples themselves, they are treated as training examples, and instead the *model* is materialized once all training instances have been processed.

The MADLib project [9, 22] adopts an interesting alternative approach by pushing functionality into the database, so that various data mining algorithms can run at scale inside the database engine. This is attractive because it eliminates the need to import/export data to other tools, which immediately solves many of the "plumbing" problems we've been discussing. The reasoning goes as follows: since many database analysts and data scientists are already performing analytics in SQL databases (e.g., cubing, aggregations, etc.), why not provide data mining capabilities in the form of familiar `SELECT... FROM... WHERE...` commands? Bringing analytics inside the database has the added advantage of letting the database engine do what it's good at: query optimization and scalable execution. Currently, MADLib (v0.3) provides supervised learning algorithms such as logic regression and decision trees as well as unsupervised algorithms such as k-means clustering and SVD matrix factorization.

Both our and the MADLib approach are similar in that they attempt to streamline data mining by deep integration—the best way to solve impedance mismatch issues from different systems is to remove the interface completely! We do so by bringing machine learning inside Pig and Hadoop; MADLib does the same via SQL with an RDBMS. The primary difference we see is the profile of the target user. In the case of MADLib, the ideal user is a database analyst who spends a lot of time working with SQL already—the project aims to augment analysts' toolbox with data mining and machine learning algorithms. In our case, Twitter data scientists are just as comfortable writing R scripts for complex data analysis as they are writing page-long SQL queries, with probably a slight preference for the former—thus, Pig's step-by-step dataflow specification comes naturally. In any case, since Pig is already established as the most widely used large-scale data processing tool at Twitter, integration of machine learning was the next logical step.

This integration-based approach to large-scale machine learning and data mining is by no means limited to MADLib and our work. For example, SystemML [16] attempts to create an environment similar to the R language. RHIPE[18] takes this one step further by attempting to parallelize R directly via Hadoop integration. This line of work is related to attempts at developing custom domain-specific languages (DSLs) for machine learning (see, for example, several examples at the NIPS 2011 Workshop on "Big Learning"). Collectively, these approaches remain immature, and we do not believe that any can be definitively recommended as best practice (including ours). The community is still in a phase of experimentation, but the substantial interest in large-scale data mining frameworks will undoubtedly lead to interesting innovations in the future.

We conclude this section by identifying two underexplored questions deserving of attention. First, we believe that visualization is an important aspect of big data mining, both from the perspective of communicating results (i.e., "telling a story" to one's peers, managers, etc.) and as a vehicle for generating insights (i.e., visualizations that help the data scientist learn about the problem). As an example, see Rios and Lin [47] for a few case studies that we recently shared. Browser-based toolkits such as d3.js [5] have emerged as the preferred method of creating and sharing visualizations, which means that development is, for the most part, limited to an engineer's laptop—this brings us back to the awkwardness of generating and distilling data using Pig, then copying data out of HDFS. Furthermore, a browser's ability to deal with large datasets is limited—in practice, a few megabytes at the most—which implies either a large degree of sampling or pre-aggregation along certain dimensions. Constraints on data preparation limit the extent to which visualizations can generate insights—it is impossible to identify the impact of features that have been discarded or rolled up in aggregate. Although we are aware of work to provide visualization support for machine learning [44], much more remains to be done. We have recently read about Facebook's use of Tableau,[19] which is clearly an attempt at solving this problem, but we were not able to find sufficient technical details to comment on the architecture.

The second open problem concerns real-time (or at least rapid) interactions with large datasets. Since data mining is an iterative process, we desire tools that shorten the development cycle and enable faster experimentation. This is a capability that our Hadoop-based stack currently cannot provide. Here is the scenario that our data scientists deal with on a daily basis: Write a Pig script and submit a job. Wait five minutes for the job to finish. Discover that the output is empty because of the wrong join key. Fix simple bug. Resubmit. Wait another five minutes. Rinse, repeat. It's fairly obvious that these long debug cycles hamper productivity; this is a fundamental weakness of MapReduce as a batch system. Examples of recent systems that are designed to enable low-latency analytics include Dremel [40], Spark [52], PowerDrill [19], and Impala,[20] although the general problem is far from solved.

6. CONCLUSIONS

Rather than re-enumerating all the best practices, recommendations, and unresolved questions in this paper, we conclude with a discussion of where we feel the field is going from a different perspective. At the end of the day, an efficient and successful big data analytics platform is about achieving the right balance between several competing factors: speed

[18] http://www.datadr.org/

[19] https://twitter.com/tableau/status/266974538175242240

[20] https://github.com/cloudera/impala

of development, ease of analytics, flexibility, scalability, robustness, etc. For example, a small team might be able to iterate on the front-end faster with JSON logging, eschewing the benefits of having schemas, but experience tells us that the team is accruing technical debt in terms of scalability challenges down the road. Ideally, we would like an analytics framework that provides all the benefits of schemas, data catalogs, integration hooks, robust data dependency management and workflow scheduling, etc. while requiring *zero* additional overhead. This is perhaps a pipe dream, as we can point to plenty of frameworks that have grown unusable under their own weight, with pages of boilerplate code necessary to accomplish even a simple task. How to provide useful tools while staying out of the way of developers is a difficult challenge.

As an organization grows, the analytics infrastructure will evolve to reflect different balances between various competing factors. For example, Twitter today generally favors scale and robustness over sheer development speed, as we know that if things aren't done "right" to begin with, they'll become maintenance nightmares down the road. Previously, we were more focused on just making things possible, implementing whatever expedient was necessary. Thinking about the evolution of analytics infrastructure in this manner highlights two challenges that merit future exploration:

First, are there prototypical evolutionary stages that data-centric organizations go through? This paper shares the Twitter experience as a case study, but can we move beyond anecdotes and war stories to a more formal classification of, for example, *Stage-I*, *Stage-II*, ... *Stage-N* organizations? In this hypothetical taxonomy, each stage description would be accompanied by the most pressing challenges and specific recommendations for addressing them. We would like to formalize the advice presented in this paper in terms of specific contexts in which they are applicable.

Second, how do we smoothly provide technology support that will help organizations grow and transition from stage to stage? For example, can we provide a non-disruptive migration path from JSON logs to Thrift-based logs? How do we provide support for deep integration of predictive analytics down the road, even though the organization is still mostly focused on descriptive aggregation-style queries today? If a framework or set of best practices can provide a smooth evolutionary path, then it may be possible for an organization to optimize for development speed early on and shift towards scale and robustness as it matures, avoiding disruptive infrastructure changes in the process.

In this paper, we have attempted to describe what it's like "in the trenches" of a production big data analytics environment. We hope that our experiences are useful for both practitioners and academic researchers. Practitioners should get a few chuckles out of our war stories and know how to avoid similar mistakes. For academic researchers, a better understanding of the broader context of big data mining can inform future work to streamline insight generation activities. We'd like to think that this paper has, however slightly, contributed to the community's shared knowledge in building and running big data analytics infrastructure.

7. ACKNOWLEDGMENTS

None of the work described here would have been possible without the analytics infrastructure engineers and data scientists at Twitter: this paper distills their collective wisdom, successes, and frustrations. We'd like to thank Albert Bifet and Wei Fan for the invitation to write this article, and Joe Hellerstein and Anthony Tomasic for helpful comments on a previous draft. The first author is grateful to Esther and Kiri for their loving support and dedicates this work to Joshua and Jacob.

8. REFERENCES

[1] A. Agarwal, O. Chapelle, M. Dudik, and J. Langford. A reliable effective terascale linear learning system. In *arXiv:1110.4198v1*, 2011.

[2] M. Banko and E. Brill. Scaling to very very large corpora for natural language disambiguation. In *ACL*, 2001.

[3] J. Basilico, M. Munson, T. Kolda, K. Dixon, and W. Kegelmeyer. COMET: A recipe for learning and using large ensembles on massive data. In *ICDM*, 2011.

[4] R. Bekkerman and M. Gavish. High-precision phrase-based document classification on a modern scale. In *KDD*, 2011.

[5] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-Driven Documents. In *InfoVis*, 2011.

[6] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT*, 2010.

[7] T. Brants, A. Popat, P. Xu, F. Och, and J. Dean. Large language models in machine translation. In *EMNLP*, 2007.

[8] E. Chang, H. Bai, K. Zhu, H. Wang, J. Li, and Z. Qiu. PSVM: Parallel Support Vector Machines with incomplete Cholesky factorization. In *Scaling up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, 2012.

[9] J. Cohen, B. Dolan, M. Dunlap, J. Hellerstein, and C. Welton. MAD skills: New analysis practices for big data. In *VLDB*, 2009.

[10] G. Cormack, M. Smucker, and C. Clarke. Efficient and effective spam filtering and re-ranking for large web datasets. In *arXiv:1004.5168v1*, 2010.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[12] J. Dean and S. Ghemawat. MapReduce: A flexible data processing tool. *CACM*, 53(1):72–77, 2010.

[13] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). In *VLDB*, 2010.

[14] C. Dyer, A. Cordova, A. Mont, and J. Lin. Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce. In *StatMT Workshop*, 2008.

[15] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of MapReduce: The Pig experience. In *VLDB*, 2009.

[16] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *ICDE*, 2011.

[17] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V. Ye. Building LinkedIn's

real-time activity data pipeline. *Bulletin of the Technical Committee on Data Engineering*, 35(2):33–45, 2012.

[18] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[19] A. Hall, O. Bachmann, R. Büssow, S. Gănceanu, and M. Nunkesser. Processing a trillion cells per mouse click. In *VLDB*, 2012.

[20] J. Hammerbacher. Information platforms and the rise of the data scientist. In *Beautiful Data: The Stories Behind Elegant Data Solutions*. O'Reilly, 2009.

[21] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *ICDE*, 2011.

[22] J. Hellerstein, C. Ré, F. Schoppmann, D. Wang, E. Fratkin, A. Gorajek, K. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library or MAD skills, the SQL. In *VLDB*, 2012.

[23] T. Hey, S. Tansley, and K. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, Washington, 2009.

[24] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. Trojan data layouts: Right shoes for a running elephant. In *SoCC*, 2011.

[25] T. Joachims, L. Granka, B. Pan, H. Hembrooke, F. Radlinski, and G. Gay. Evaluating the accuracy of implicit feedback from clicks and query reformulations in Web search. *ACM TOIS*, 25(2):1–27, 2007.

[26] I. Kanaris, K. Kanaris, I. Houvardas, and E. Stamatatos. Words versus character n-grams for anti-spam filtering. *International Journal on Artificial Intelligence Tools*, 16(6):1047–1067, 2007.

[27] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. In *VAST*, 2012.

[28] R. Kohavi, A. Deng, B. Frasca, R. Longbotham, T. Walker, and Y. Xu. Trustworthy online controlled experiments: Five puzzling outcomes explained. In *KDD*, 2012.

[29] R. Kohavi, R. Henne, and D. Sommerfield. Practical guide to controlled experiments on the web: Listen to your customers not to the HiPPO. In *KDD*, 2007.

[30] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *NetDB*, 2011.

[31] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy. The unified logging infrastructure for data analytics at Twitter. In *VLDB*, 2012.

[32] H. Li. *Learning to Rank for Information Retrieval and Natural Language Processing*. Morgan & Claypool, 2011.

[33] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[34] J. Lin and A. Kolcz. Large-scale machine learning at Twitter. In *SIGMOD*, 2012.

[35] J. Lin, D. Ryaboy, and K. Weil. Full-text indexing for optimizing selection operations in large-scale data analytics. In *MAPREDUCE Workshop*, 2011.

[36] Y. Lin, D. Agrawal, C. Chen, B. Ooi, and S. Wu. Llama: Leveraging columnar storage for scalable join processing in the MapReduce framework. In *SIGMOD*, 2011.

[37] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[38] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In *NIPS*, 2009.

[39] R. McDonald, K. Hall, and G. Mann. Distributed training strategies for the structured perceptron. In *HLT*, 2010.

[40] S. Melnik, A. Gubarev, J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *VLDB*, 2010.

[41] A. Ng, G. Bradski, C.-T. Chu, K. Olukotun, S. Kim, Y.-A. Lin, and Y. Yu. Map-Reduce for machine learning on multicore. In *NIPS*, 2006.

[42] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[43] B. Panda, J. Herbach, S. Basu, and R. Bayardo. MapReduce and its application to massively parallel learning of decision tree ensembles. In *Scaling up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, 2012.

[44] K. Patel, N. Bancroft, S. Drucker, J. Fogarty, A. Ko, and J. Landay. Gestalt: Integrated support for implementation and analysis in machine learning. In *UIST*, 2010.

[45] D. Patil. *Building Data Science Teams*. O'Reilly, 2011.

[46] D. Patil. *Data Jujitsu: The Art of Turning Data Into Product*. O'Reilly, 2012.

[47] M. Rios and J. Lin. Distilling massive amounts of data into simple visualizations: Twitter case studies. In *Workshop on Social Media Visualization at ICWSM*, 2012.

[48] D. Sculley, M. Otey, M. Pohl, B. Spitznagel, J. Hainsworth, and Y. Zhou. Detecting adversarial advertisements in the wild. In *KDD*, 2011.

[49] K. Svore and C. Burges. Large-scale learning to rank using boosted decision trees. In *Scaling up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, 2012.

[50] B. Taylor, D. Fingal, and D. Aberdeen. The war against spam: A report from the front line. In *NIPS Workshop on Machine Learning in Adversarial Environments*, 2007.

[51] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at Facebook. In *SIGMOD*, 2010.

[52] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.