Goods: Organizing Google's Datasets

Alon Halevy[2*], Flip Korn[1], Natalya F. Noy[1], Christopher Olston[1], Neoklis Polyzotis[1], Sudip Roy[1], Steven Euijong Whang[1]

[1]Google Research [2]Recruit Institute of Technology

alon@recruit.ai, {flip, noy, olston, npolyzotis, sudipr, swhang}@google.com

ABSTRACT

Enterprises increasingly rely on structured datasets to run their businesses. These datasets take a variety of forms, such as structured files, databases, spreadsheets, or even services that provide access to the data. The datasets often reside in different storage systems, may vary in their formats, may change every day. In this paper, we present Goods, a project to rethink how we organize structured datasets at scale, in a setting where teams use diverse and often idiosyncratic ways to produce the datasets and where there is no centralized system for storing and querying them. Goods extracts metadata ranging from salient information about each dataset (owners, timestamps, schema) to relationships among datasets, such as similarity and provenance. It then exposes this metadata through services that allow engineers to find datasets within the company, to monitor datasets, to annotate them in order to enable others to use their datasets, and to analyze relationships between them. We discuss the technical challenges that we had to overcome in order to crawl and infer the metadata for billions of datasets, to maintain the consistency of our metadata catalog at scale, and to expose the metadata to users. We believe that many of the lessons that we learned are applicable to building large-scale enterprise-level data-management systems in general.

1. INTRODUCTION

Most large enterprises today witness an explosion in the number of datasets that they generate internally for use in ongoing research and development. The reason behind this explosion is simple: by allowing engineers and data scientists to consume and generate datasets in an unfettered manner, enterprises promote fast development cycles, experimentation, and ultimately innovation that drives their competitive edge. As a result, these internally generated datasets often become a prime asset of the company, on par with source code and internal infrastructure. However, while enterprises have developed a strong culture on how to manage the latter, with source-code development tools and methodologies that we now consider "standard" in the industry (e.g., code versioning and indexing, reviews, or testing), similar approaches do not generally exist for managing datasets. We argue that developing principled and flexible approaches to dataset management has become imperative, lest companies run the risk of internal siloing of datasets, which, in turn, results in significant losses in productivity and opportunities, duplication of work, and mishandling of data.

Enterprise Data Management (EDM) is one common way to organize datasets in an enterprise setting. However, in the case of EDM, stakeholders in the company must embrace this approach, using an EDM system to publish, retrieve, and integrate their datasets. An alternative approach is to enable complete freedom within the enterprise to access and generate datasets and to solve the problem of finding the right data in a post-hoc manner. This approach is similar in spirit to the concept of *data lakes* [4, 22], where the lake comprises and continuously accumulates all the datasets generated within the enterprise. The goal is then to provide methods to "fish" the right datasets out of the lake on the as-needed basis.

In this paper, we describe Google Dataset Search (Goods), such a post-hoc system that we built in order to organize the datasets that are generated and used within Google. Specifically, Goods collects and aggregates metadata about datasets after the datasets were created, accessed, or updated by various pipelines, without interfering with dataset owners or users. Put differently, teams and engineers continue to generate and access datasets using the tools of their choice, and Goods works in the background, in a non-intrusive manner, to gather the metadata about datasets and their usage. Goods then uses this metadata to power services that enable Google engineers to organize and find their datasets in a more principled manner.

Figure 1 shows a schematic overview of our system. Goods continuously crawls different storage systems and the production infrastructure (e.g., logs from running pipelines) to discover which datasets exist and to gather metadata about each one (e.g., owners, time of access, content features, accesses by production pipelines). Goods aggregates this metadata in a central catalog and correlates the metadata about a specific dataset with information about other datasets.

Goods uses this catalog to provide Google engineers with services for dataset management. To illustrate the types of services powered by Goods, imagine a team that is responsible for developing natural language understanding (NLU) of text corpora (say, news articles). The engineers on the team may be distributed across the globe and they maintain several pipelines that add annotations to different text corpora. Each pipeline can have multiple stages that add annotations based on various techniques including phrase chunking, part-of-speech tagging, and co-reference resolution. Other teams can consume the datasets that the NLU team generates, and the NLU team's pipelines may consume datasets from other teams.

*Work done while at Google Research.

SIGMOD/PODS'16 June 26 - July 01, 2016, San Francisco, CA, USA

ACM ISBN 978-1-4503-3531-7/16/06.

DOI: http://dx.doi.org/10.1145/2882903.2903730

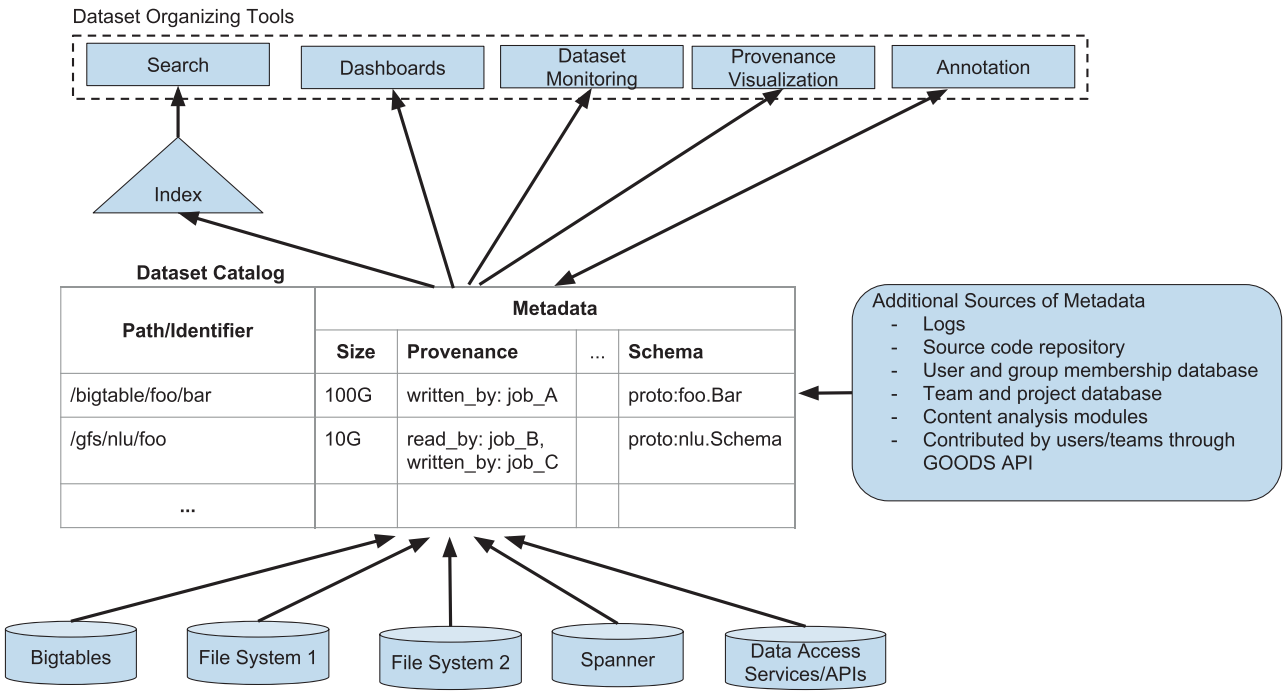

Figure 1: Overview of Google Dataset Search (GOODS). The figure shows the GOODS dataset catalog that collects metadata about datasets from various storage systems as well as other sources. We also infer metadata by processing additional sources such as logs and information about dataset owners and their projects, by analyzing content of the datasets, and by collecting input from the GOODS users. We use the information in the catalog to build tools for search, monitoring, and visualizing flow of data.

Based on the information in its catalog, GOODS provides a *dashboard* for the NLU team (in this case, dataset *producers*), which displays all their datasets and enables browsing them by facets (e.g., owner, data center, schema). Even if the team's datasets are in diverse storage systems, the engineers get a unified view of all their datasets and dependencies among them. GOODS can *monitor* features of the dataset, such as its size, distribution of values in its contents, or its availability, and then alert the owners if the features change unexpectedly.

Another important piece of information that GOODS provides is the dataset *provenance*: namely, the information about which datasets were used to create a given dataset (upstream datasets), and those that rely on it (downstream datasets). Note that both the upstream and downstream datasets may be created by other teams. When an engineer in the NLU team observes a problem with a dataset, she can examine the provenance visualization to determine whether a change in some upstream dataset had caused the problem. Similarly, if the team is about to make a significant change to its pipeline or has discovered a bug in an existing dataset that other teams have consumed already, they can quickly notify those affected by the problem.

From the perspective of dataset *consumers*, such as those not part of the NLU team in our example, GOODS provides a *search engine* over all the datasets in the company, plus facets for narrowing search results, to find the most up-to-date or potentially important datasets. GOODS presents a profile page for every dataset, which helps users unfamiliar with the data to understand its schema and to create boilerplate code to access and query the data. The profile page also contains the information on datasets with content similar to the content of the current dataset. The similarity information may enable novel combinations of datasets: for example, if two datasets share a primary key column, then they may provide complementary information and are therefore a good candidate for joining.

GOODS allows users to expand the catalog with crowd-sourced metadata. For instance, dataset owners can *annotate* datasets with descriptions, in order to help users figure out which datasets are appropriate for their use (e.g., which analysis techniques are used in certain datasets and which pitfalls to watch out for). Dataset auditors can tag datasets that contain sensitive information and alert dataset owners or prompt a review to ensure that the data is handled appropriately. In this manner, GOODS and its catalog become a hub through which users can share and exchange information about the generated datasets. GOODS also exposes an API through which teams can contribute metadata to the catalog both for the teams own restricted use as well as to help other teams and users understand their datasets easily.

As we discuss in the rest of the paper, we addressed many challenges in designing and building GOODS, arising from the sheer number of datasets (tens of billions in our case), the high churn in terms of updates, the sizes of individual datasets (gigabytes or terabytes in many cases), the many different data formats and stores they reside in, and the varying quality and importance of information collected about each dataset. Many of the challenges that we addressed in GOODS were precipitated by the scale and characteristics of the data lake at Google. However, we believe that our experience and the lessons that we learned will apply to similar systems in other enterprises.

2. CHALLENGES

In this section, we describe in more detail the challenges that we addressed in building Goods. While some of these challenges are specific to Google's setup, we believe that most of the following points carry over to other large enterprises.

2.1 Scale of the number and size of datasets

While we anticipated the existence of a large number of datasets in the company, the actual count far exceeded our initial calculations. The current catalog indexes over 26 billion datasets even though it includes only those datasets whose access permissions make them readable by *all* Google engineers. We expect that the catalog will more than double in the number of datasets when we index the datasets with restricted access permissions and when we start supporting other storage systems. It is important to note that the catalog already excludes many types of uninteresting datasets (e.g., we discard known "marker" files that are content free) and normalizes paths to avoid obvious redundancies (e.g., we normalize paths corresponding to different shards of the same dataset to a common path and do not store them separately in the catalog).

At this scale, gathering metadata for all datasets becomes infeasible. Indeed, even if we spend one second per dataset (and many of the datasets are too large to process in one second), going through a catalog with 26 billion datasets using a thousand parallel machines still requires around 300 days. Thus, we must develop strategies that prioritize and optimize processing of the datasets.

The scale issue is aggravated by "n-square" problems that arise in metadata inference. For instance, Goods identifies datasets that contain similar or identical content, both overall as well as for individual columns. Comparing any two datasets to each other can already be expensive due to large dataset sizes, but a naïve pairwise comparison of contents of billions of datasets is infeasible.

2.2 Variety

Datasets are stored in many formats (text files, csv files, Bigtables [13], etc.) and storage systems (GoogleFS, database servers, etc.), each with its own type of metadata and access characteristics. This diversity makes it difficult to define a single "dataset" concept that covers all actual dataset types. Hiding this variety and complexity from the users and presenting a uniform way to access and query information about all types of datasets is both a goal and a challenge for Goods.

Even more important is the variety in the cost of metadata extraction, which can change vastly depending on the type and size of the dataset and the type of metadata. Therefore, our metadata-extraction process needs to be differential: we must identify which datasets are important to cover, and then to perform metadata inference based on the cost and benefit for having the particular type of metadata.

Variety also manifests in the relationships among datasets, which in turn affect how we model and store metadata in the catalog. Take for instance a Bigtable dataset [13]. It comprises several column families, with each family inheriting metadata from the containing Bigtable but also having its own metadata and access properties. As a result, we can view column families both as individual datasets and as part of the overall Bigtable dataset. Furthermore, the underlying storage infrastructure for a Bigtable is provided by a distributed file system, and so we can also view the corresponding files as datasets. In the case of Bigtable, the decision to hide these underlying files in favor of the Bigtable dataset seems reasonable. However, a similar decision is less clear in other cases. For instance, we include database tables (specifically, Dremel tables [19]) in our catalog. These tables were created from other files (also in our catalog), and in this case it is meaningful to have both the files and the database tables as separate (but connected) datasets in the catalog given that their access patterns are sufficiently different. Note that this last example illustrates a type of dataset aliasing. Aliases can arise in several ways in our catalog, and we have dealt with each alias type separately depending on the corresponding usage patterns.

Number of datasets	26 billion
Number of paths added per day	1.6 billion
Number of paths deleted per day	1.6 billion
Number of storage systems	6
Number of dataset formats	> 20

Table 1: The scale, variety, and churn of the entries in the Goods catalog.

2.3 Churn of the catalog entries

Every day, production jobs generate new datasets, and old datasets get deleted—either explicitly or because their designated time-to-live (TTL) has expired. In fact, we found that more than 5% (i.e., about 1 billion) of the datasets in the catalog are deleted every day. Almost an equal number of new entries are added back as well. This level of churn adds more considerations to how we prioritize for which datasets we compute the metadata and which datasets we include in the catalog. For example, many of the datasets that have a limited Time-To-Live (TTL) from their creation are intermediate results of a large production pipeline that are garbage collected after a few days. One possibility is to ignore these transient datasets for metadata extraction, or even to exclude them from the catalog. However, there are two considerations. First, some of these datasets have long TTLs (e.g., measured in weeks) and hence their value to users can be high when the datasets are just created. Second, as we discuss later, some of these transient datasets link non-transient datasets to each other and are thus critical in the computation of dataset provenance. Hence, entirely blacklisting transient datasets was not an option for Goods.

2.4 Uncertainty of metadata discovery

Because Goods explicitly identifies and analyzes datasets in a post-hoc and non-invasive manner, it is often impossible to determine all types of metadata with complete certainty. For instance, many datasets consist of records whose schema conforms to a specific protocol buffer [23] (i.e., a nested-relational schema). However, the dataset itself does not reference the specific protocol buffer that describes its content—the association between a protocol buffer and a dataset is implicit in the source code that accesses the dataset. Goods tries to uncover this implicit association through several signals: For instance, we "match" the dataset contents against all registered types of protocol buffers within Google, or we consult usage logs that may have recorded the actual protocol buffer. This inference is inherently ambiguous and can result in several possible matches between a dataset and protocol buffers.

Similarly, Goods analyzes the datasets to determine which fields could serve as primary keys, but relies on an approximate process in order to deal with the scale of the problem—another uncertain inference.

Most of this uncertainty arises because we process the datasets in a post-hoc fashion: we do not require dataset owners to change their workflows in order to associate this type of metadata with the datasets when the owners create the datasets. Instead, we opt to collect dataset metadata that is already logged in different corners

Metadata Groups	Metadata
Basic	size, format, aliases, last modified time, access control lists
Content-based	schema, number of records, data fingerprint, key field, frequent tokens, similar datasets
Provenance	reading jobs, writing jobs, downstream datasets, upstream datasets
User-supplied	description, annotations
Team and Project	project description, owner team name
Temporal	change history

Table 2: Metadata in the Goods catalog.

of the existing infrastructure and then we aggregate and clean the metadata for further usage.

2.5 Computing dataset importance

After we discover and catalog the datasets, reasoning about their relative importance to the users presents additional challenges. To start with, the basic question of what makes a dataset important is hard to answer. Looking at a dataset in isolation can provide some hints, but it is often necessary to examine the dataset in a more global context (e.g., consider how often production pipelines access the dataset) in order to understand its importance.

Note that the notion of importance—and relative importance—comes up prominently in the context of Web search. However, ranking and importance among structured datasets in an enterprise setting is significantly different from the Web search setting: the only explicit links that we have are the provenance links, which do not necessarily signify importance. Furthermore, many of the signals that we can use for Web search (e.g., anchor text) do not exist for datasets, whereas datasets can provide structured context that the Web pages do not have.

In addition to the importance of datasets to the users, a different notion of importance comes up when we prioritize the datasets for which we derive metadata. For example, we would often consider transient datasets to be unimportant. However, if a transient dataset provides a provenance link between non-transient, important datasets, then it is natural to boost its importance accordingly.

2.6 Recovering dataset semantics

Understanding the semantics of dataset contents is extremely useful in searching, ranking, and describing the datasets. Suppose that we know the schema of a dataset and that some field in the schema takes integer values. Now, suppose that through some inference on the dataset contents we are able to identify that these integer values are IDs of known geographic landmarks. We can use this type of content semantics to improve search when a user searches Goods for geographic data. In general, by lifting the level of abstraction from raw bytes to concepts we can make inferences that lead to deeper and cleaner dataset metadata. However, identifying the semantics from raw data is a hard problem even for small datasets [12] because there is rarely enough information in the data to make this inference. Performing such inference for datasets with billions of records becomes even harder.

3. THE GOODS CATALOG

Having described the challenges that we addressed in building Goods, we now turn our attention to the details of the system. We begin by taking a closer look at the Goods catalog, which lies in the core of our system. At a high level, the catalog contains an entry for each dataset that Goods discovers by crawling different storage systems within Google. While each independent storage system within the company may maintain a catalog over datasets that it serves, each such catalog has different types of metadata and data often flows from one system to another in an unfettered fashion. This freedom makes it difficult to obtain a global and unified view of the datasets available throughout the company. The Goods catalog fills this gap and is thus an important contribution in itself, even if we do not consider the services at the top of Figure 1.

The Goods catalog not only contains the individual datasets that we collect by crawling the different storage systems, but also groups related datasets into clusters, which become first-class entries in the catalog. Consider for example a dataset where a new version is produced daily, or even hourly. Our catalog will contain an entry for each version of such a dataset. However, users would often want to think of these versions as a single logical dataset. Furthermore, all these versions are likely to have some common metadata (e.g., owners or schema) and thus collecting metadata separately for each version is wasteful—and often prohibitive—in terms of resources. For these two reasons, we organize these related datasets in a cluster, which becomes a separate entry in the catalog. Goods surfaces this cluster to users as a logical dataset that represents all the generated versions. Goods also uses the cluster to optimize the computation of metadata under the assumption that all datasets in the cluster have similar properties.

In this section, we first describe the types of metadata that we associate with each dataset (Section 3.1) and then describe our mechanism for metadata extraction based on dataset clusters (Section 3.2).

3.1 Metadata

Goods bootstraps its catalog by crawling Google's storage systems (e.g., GoogleFS, Bigtable, database servers) in order to discover what datasets exist and to obtain some basic metadata such as size, owners, readers, and access permissions of datasets. However, most storage systems do not keep track of other important metadata, such as jobs that produced a dataset, teams and users that access it, its schema, and others (Table 2). This information is spread across logs written by processes that access these datasets, is encoded within the datasets themselves, or can be deduced by analyzing the content of the datasets. Thus, in addition to crawling, we perform metadata inference. In what follows, we describe in more detail the different types of metadata (collected and inferred) in the Goods catalog.

Basic metadata - The basic metadata for each dataset includes its timestamp, file format, owners, and access permissions. We obtain this basic metadata by crawling the storage systems and this process usually does not necessitate any inference. Other Goods modules often depend on this basic information to determine their behavior. For example, some of the modules bypass catalog entries that have restricted access or the entries that have not been modified recently.

Provenance - Datasets are produced and consumed by code. This code may include analysis tools that are used to query datasets, serving infrastructures that provide access to datasets through APIs, or ETL pipelines that transform it into other datasets. Often, we can understand a dataset better through these surrounding pieces of software that produce and use it. Moreover, this information helps in tracking how data flows through the enterprise as well as across

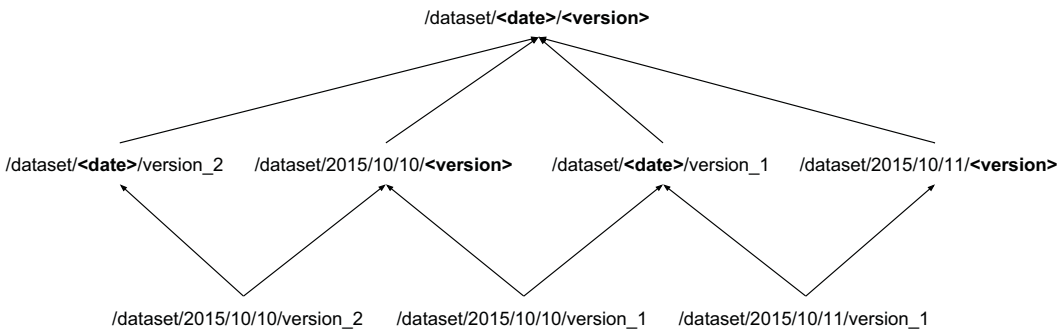

Figure 2: An example of abstraction semi-lattice

boundaries of teams and organizations within the company. Therefore, for each dataset, the Goods catalog maintains the provenance of how the dataset is produced, how it is consumed, what datasets this dataset depends on, and what other datasets depend on this dataset. We identify and populate the provenance metadata through an analysis of production logs, which provide information on which jobs read and write each dataset. We then create a transitive closure of this graph connecting datasets and jobs, in order to determine how the datasets themselves are linked to one another. For instance if a job J reads dataset D_1 and produces dataset D_2, then the metadata for D_1 contains D_2 as one of its downstream datasets and vice versa. We also take into account timing information in order to determine the earliest and latest points in time when these dependencies existed. However, the number of data-access events in the logs can be extremely high and so can be the size of the transitive closure. Therefore, we trade off the completeness of the provenance associations for efficiency by processing only a sample of data-access events from the logs and also by materializing only the downstream and upstream relations within a few hops as opposed to computing the true transitive closure.

Schema - Schema is another core type of metadata that helps us understand a dataset. The most commonly used dataset formats in Google are not self-describing, and we must infer the schema. Nearly all records within structured datasets at Google are encoded as serialized protocol buffers [23]. The difficulty lies in determining which protocol buffer was used to encode records in a given dataset. Protocol buffers that the majority of Google's datasets use are nearly always checked into Google's central code repository. Thus, we have a full list of them available to match against datasets that we have crawled. We perform this matching by scanning a few records from the file, and going through each protocol message definition to determine whether it could conceivably have generated the bytes we see in those records. Protocol buffers encode multiple logical types as the same physical type, notably string and nested messages are both encoded as variable-length byte strings. Consequently, the matching procedure is speculative and can produce multiple candidate protocol buffers. All the candidate protocol buffers, along with heuristic scores for each candidate, become part of the metadata.

Content summary - For each dataset that we are able to open and scan, we also collect metadata that summarizes the content of the dataset. We record frequent tokens that we find by sampling the content. We analyze some of the fields to determine if they contain keys for the data, individually or in combination. To find potential keys, we use the HyperLogLog algorithm [15] to estimate cardinality of values in individual fields and combinations of fields and we compare this cardinality with the number of records to find potential keys. We also collect fingerprints that have checksums for the individual fields and locality-sensitive hash (LSH) values for the content. We use these fingerprints to find datasets with content that is similar or identical to the given dataset, or columns from other datasets that are similar or identical to columns in the current dataset. We also use the checksums to identify which fields are populated in the records of the dataset.

User-provided annotations - We enable dataset owners to provide text descriptions of their datasets. These descriptions are critical to our ranking, and also help us filter out datasets that are experimental or that we should not show to the users.

Semantics - Goods combines several noisy signals in order to derive metadata about dataset semantics. For datasets whose schema conforms to a protocol buffer, Goods can examine the source code that defines the protocol buffer and extract any of the attached comments. These comments are often of high quality and their lexical analysis can provide short phrases that capture the semantics of the schema. As an actual example, some of the datasets in the Goods catalog conform to a protocol buffer with a field cryptically named `mpn`. However, the source code contains the comment "`//Model Product Number`" above the field, which disambiguates its semantics. Goods can also examine the dataset content and match it against Google's Knowledge Graph [11], to identify entities (e.g., locations, businesses) that appear in different fields.

In addition to the metadata listed above, we collect identifiers for teams that own the dataset, a description of the project to which a dataset belongs, and the history of the changes to the metadata of the dataset.

Finally, our infrastructure (Section 4) allows other teams to add their own types of metadata. For instance, a team may rely on different types of content summaries, or provide additional provenance information. The Goods catalog is becoming the unifying place for teams to collect their metadata and to access it.

3.2 Organizing datasets into clusters

The 26B datasets in the Goods catalog are not completely independent from one another. We often see different versions of a dataset that are being generated on a regular basis, datasets that are replicated across different data centers, datasets that are sharded into smaller datasets for faster loading, and so on. If we can identify natural clusters to which datasets belong, then not only we can provide the users with a useful logical-level abstraction that groups together these different versions but we can also save on metadata extraction, albeit potentially at the cost of precision. That is, instead of collecting expensive metadata for each individual dataset, we can collect metadata only for a few datasets in a cluster. We can then propagate the metadata across the other datasets in the cluster. For instance, if the same job generates versions of a dataset daily, these datasets are likely to have the same schema. Thus, we do not

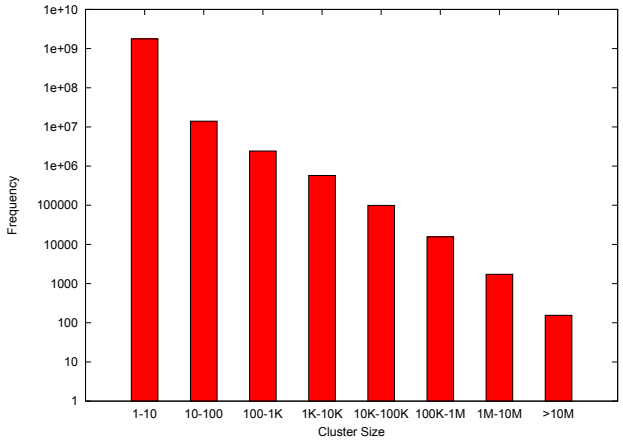

Figure 3: Distribution of cluster sizes. The X axis is the number of datasets in a cluster (cluster size). The Y axis is the number of clusters of the corresponding cluster size.

need to infer the schema for each version. Similarly, if a user provides a description for a dataset, it usually applies to all members of the cluster and not just the one version. When the clusters are large, the computational savings that we obtain by avoiding analysis of each member of the cluster can be significant.

For clustering to reduce the computational overhead, the clustering itself should be cheap. Clustering techniques that require investigating the contents of a dataset can overshadow the computational savings that we obtain by avoiding repetitive metadata extraction. Fortunately, the paths of the datasets often give hints on how to cluster them via embedded identifiers for timestamps, versions, and so on. For example, consider a dataset that is produced daily and let `/dataset/2015-10-10/daily_scan` be the path for one of its instances. We can *abstract* out the day portion of the date to get a generic representation of all datasets produced in a month: `/dataset /2015-10-<day>/daily_scan`, representing all instances from October 2015. By abstracting out the month as well, we can go up the hierarchy to create abstract paths that represent all datasets produced in the same year: `/dataset/2015-<month>-<day>/daily_scan`.

By composing hierarchies along different dimensions, we can construct a *granularity semi-lattice* structure where each node corresponds to a different *granularity* of viewing the datasets. Figure 2 shows an example of such a semi-lattice obtained by composing two hierarchies—one along date and the other along version number.

Table 3 lists the abstraction dimensions that we currently use to construct the granularity semi-lattice for each dataset. By abstracting out the various dimensions from all dataset paths, we eventually obtain a set of semi-lattices whose non-leaf nodes represent different choices for grouping datasets into clusters. We can optimize the selection of clusters with a suitable objective function over the current state of the catalog, but the daily catalog churn can cause a frequent recomputation of clusters. Therefore, users may see different clusters (representing logical datasets) from day to day, which can be confusing. We adopted a simple solution that works well in practice: we create an entry only for the top-most element of each semi-lattice. Going back to the example of Figure 2, the catalog would have an entry for the cluster `/dataset/<date>/<version>`, representing the three datasets at the bottom of the lattice. This approach keeps the number of cluster entries low, guarantees that each dataset maps to exactly one cluster, and maintains a stable set of clusters over time.

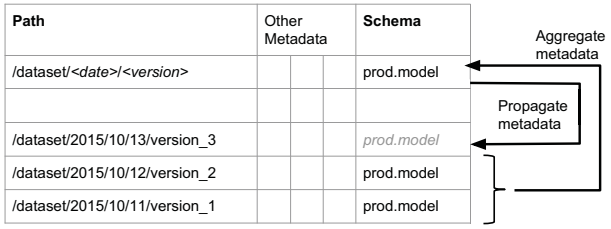

Figure 4: An example of propagating the *owners* metadata to an unanalyzed dataset through the cluster's representative element.

After we compute the clusters, we obtain the metadata for each cluster by aggregating the metadata for individual members. For instance, if we know the schema for several members of the cluster, and it is the same for all of them, we can propagate this schema for the cluster as a whole (Figure 4). Whether to materialize this propagated information or simply to compute it on-demand is an application-specific design decision. In our case, we compute it on demand to differentiate explicitly between propagated metadata and metadata that we obtained through real analysis.

Figure 3 shows a distribution of the number of datasets within each cluster in our catalog. The figure shows that clustering can compress dramatically the set of "physical" datasets into a much smaller set of "logical" datasets, thus making it easier for users to inspect the catalog. Moreover, the computational savings from metadata smearing are significant, particularly for the extremely large clusters.

4. BACKEND IMPLEMENTATION

In this section, we focus on the implementation details of building and maintaining the catalog that we described in Section 3. We discuss the physical structure of the catalog, our approach to scaling up the modules that populate the catalog, consistency and garbage collection, and finally, fault tolerance.

4.1 Catalog storage

We use Bigtable [13], a scalable, temporal key–value store, as the storage medium for our catalog. Each row represents one dataset or a dataset cluster (Section 3.2), with the dataset path or cluster path as the key. Bigtable offers per-row transactional consistency, which is a good fit as most (although, not all) of the processing in our system is per dataset. For instance, we can infer a schema for a dataset without looking at entries for other datasets; we can analyze the content of a dataset by examining a single row.

There are some aspects of our system that deviate from this per-dataset processing. For example, we aggregate information from multiple rows into logical datasets in our abstraction lattice (Section 3.2). Propagating metadata across datasets in the same cluster also does not conform to the model of processing each dataset independently. However, this metadata propagation is best effort and does not require strong consistency.

At the physical level, a Bigtable comprises several independent column families. We keep data that is accessed only by batch jobs (vs. served to a frontend tool) in separate column families that we tuned for batch processing (highly compressed, and not memory-resident). For instance, our largest column family contains raw provenance data that we use to compute the provenance graph but that we do not directly serve at the front end (Section 5): There, we serve the provenance information only at the level of abstracted

Abstraction Dimension	Description	Examples of paths with instances
Timestamps	All specifications of dates and times	/gfs/generated_at_**20150505T20:21:56**
Data-center Names	Specification of data center names	/gfs/**oregon**/dataset
Machine Names	Hostnames of machines (either user's or one in the data center)	/gfs/dataset/**foo.corp.google.com**
Version	Numeric and hexa-numeric version specifications	/gfs/dataset/**0x12ab12c**/bar
Universally Unique Identifier	UUIDs as specified in RFC4122[6]	/gfs/dataset/**30201010-5041-7061-9081-F0E0D0C0B0AA**/foo

Table 3: The dimensions that GOODS uses to abstract dataset paths. The examples illustrate portions of the paths that correspond to the abstraction dimensions.

dataset clusters. Thus, we can aggressively compress this large column family.

We store two kinds of metadata for each row in our Bigtable-backed catalog: (a) metadata for the datasets (Section 3); (b) metadata about the outcome of the modules that processed a given dataset, *status metadata*. The status metadata lists each module that processed a particular entry, with timestamp, success status, and error message, if any. We use the status metadata to coordinate execution of modules (Section 4.2), and for our own inspection of the system (What fraction of datasets has module X processed successfully? What is the most common error code?). In conjunction with Bigtable's temporal data model, the status metadata was also extremely useful for debugging. We configure the Bigtable to retain multiple generations of the status metadata, allowing us to see what our modules have been doing over time (e.g., on a given problematic dataset, when did the module start emitting an error? is the error deterministic?).

4.2 Batch job performance and scheduling

Our system is comprised of two types of jobs: (1) a large number of diverse batch-processing jobs; and (2) a small number of jobs that serve our front end and the API. In addition, we designed the system to be extensible and to accommodate crawlers for new sources of datasets and provenance and other metadata, and new analysis modules. Some of our batch jobs are fairly quick and typically finish a full pass over our catalog in a few hours; others, such as the ones that analyze the content of the datasets, take many days to catch up with a fresh crawl that added new datasets to the catalog. We schedule such jobs to run in geographical proximity to the datasets that they analyze, which are distributed worldwide.

We allow all the jobs to run independently from one another: we do not restrict the order in which jobs run, or constrain whether or not they run concurrently. Some of the jobs may be broken at any given time and we can take them offline temporarily. Each job includes one or more modules, such as crawlers or analyzers.

Individual modules often depend on other modules and cannot process a given dataset until some other modules processed it. For example, the module that computes fingerprints for columns needs to know a dataset's schema, and hence has a dependency on the module that determines the schema of a dataset—a so-called *Schema Analyzer* module. The modules coordinate their execution with one another at the granularity of individual Bigtable rows, using the status metadata that we mentioned earlier. If a module *A* must (successfully) process a row before module *B*, then when module *B* examines a row, it checks for a status metadata entry indicating a successful visit by module *A*. If no such status entry is present, then module *B* bypasses that row; it will try again next time it runs. If module *A* re-processes a row, then upon its next visit to the row, module *B* will also re-process it, to propagate the freshest metadata (e.g. re-fingerprint based on the updated schema). Modules also use their own status metadata to avoid re-processing rows they have already processed within a configurable freshness window.

Most jobs are configured to run daily, and finish comfortably in 24 hours. When jobs overrun their daily cycle, we optimize and/or add parallelism. These jobs use each 24-hour cycle to process new catalog rows and/or refresh already-processed ones that have fallen outside the freshness window.

After a large influx of new datasets (e.g. incorporating a new source of dataset paths), our most heavyweight job, the Schema Analyzer, takes days, or sometimes weeks, to catch up. We use a simple prioritization mechanism to ensure that our most important datasets do not get neglected by the Schema Analyzer during this "catch-up" scenario: We heuristically designate datasets having user-supplied annotations or high provenance centrality as "important," and schedule two instances of the job: one instance processes only the important datasets and can get through them quickly, and another instance aims to process all datasets but may get only to a fraction of them by day's end. In practice, as with Web crawling, ensuring good coverage and freshness for the "head" of the importance distribution is enough for most user scenarios.

Our large crawler jobs perform "blind writes" to the catalog: They read data from a source and write all of the data to our Bigtable. Bigtable does not distinguish between insertions and updates, so this approach yields a combination of no-op updates and insertions. This approach is more efficient than doing a read of our catalog to anti-join it with the new source crawl. However, we must take care to avoid no-op writes in certain cases, because it can cause dependent modules to re-run needlessly, or block garbage collection (see Section 4.4).

4.3 Fault tolerance

With such a large number and variety of datasets being analyzed, we encounter many different kinds of problems. For modules that process individual datasets, we record per-dataset errors in the status metadata of the dataset's entry in the catalog. Status metadata that indicates that a module finished with an error, triggers (a bounded number of) retries. The modules that do not process each dataset in isolation (e.g. provenance linking), rely on a job-wide status metadata entry that indicates the start time of a job and whether the job succeeded. For example, the provenance-linking module incorporates a dataset–job link into the transitive provenance graph only if the link's timestamp is more recent than the last successful execution of the module (as recorded in the module-wide status metadata). This approach is conservative: we may redo some Bigtable writes if the previous execution of the module failed part-way. However, this approach ensures that the resulting provenance graph is correct because Bigtable writes are idempotent. Furthermore, it enables us to mark the recorded job-provenance information as "consumed," a feature that becomes critical for garbage collection, as we discuss later.

Several of our modules that examine content of datasets use a variety of libraries specific to different file formats. At times, these libraries crash or go into infinite loops. (While we work to track down and eliminate these cases, it is not feasible to eradicate them entirely, especially given the evolving nature of files and file formats in our environments.) Because we cannot have long-running analysis jobs crashing or hanging, we sandbox such potentially dangerous jobs in a separate process. We then use a watchdog thread to convert long stalls into crashes while allowing the rest of the pipeline to proceed.

We replicate our catalog at multiple geographical locations. Writes go to a master and get replicated asynchronously in the background elsewhere.

4.4 Garbage collection of metadata

We ingest and create a huge volume of data daily. A nontrivial fraction of this data is transient. After we consume such transient data to build the provenance graph, we can remove the entries that correspond to the deleted datasets, as long as our modules have consumed the associated metadata to update the catalog. We initially favored a simple, conservative approach to garbage collection. For instance, we would delete a row if it had not been updated for one week. However, a few incidents in which our catalog became badly bloated taught us that aggressive garbage collection is necessary. Early in our project, on two occasions we had to disable all crawlers and non-garbage-collection-related analysis modules for several days to recover from this situation.

The garbage-collection mechanism that we have implemented currently addresses several constraints that we discovered along the way:

1. Conditions for removing rows are best expressed as declarative predicates that use both metadata and statuses of other modules that may have accessed or updated the row. For example, we have the following condition on when we can remove a dataset from the Goods catalog: "the dataset has been deleted from the storage system, and its most recently updated provenance information has been processed by a transitive provenance linker module that finished successfully."

2. When we delete an entry from a catalog, we must ensure that we do not create so-called "dangling rows": Recall that Bigtable does not differentiate between insertions and updates. Thus, when we remove a row, we must ensure that no other concurrently running module will add that row back with only partial information (specifically, only the information that the specific module is responsible for). For example, assume that there is a race condition between the garbage collector and a metadata-inference module that is examining the same row. The garbage collector removes the row and then the metadata-inference module inserts the row back; this row will contain only the information that this module infers. This sequence results in loss of information about where the corresponding dataset was crawled, possibly compromising the integrity of other modules.

3. All other module must be able to run independently of and simultaneously with the garbage collection.

Bigtable supports *conditional mutations*, which are stylized transactions that update or delete a Bigtable row if a given predicate yields true, in a transactional fashion. Having Bigtable updates from all modules to be conditioned on the row not having been deleted, proved to be too expensive: Conditional mutations incur substantial log-structured read overhead.

Our ultimate design permits all modules other than the garbage collector to perform non-transactional updates. To enable this flexibility, the garbage collection occurs in two phases: (a) In the first phase, we use declarative predicates that approve removing a catalog row (see the first condition above). However, in this first phase our garbage collector does not actually delete the Bigtable entry, but puts a special *tombstone* marker on it. (b) 24 hours later (see below for more on this threshold), if the row still meets the deletion criteria, we delete it; otherwise, we remove the tombstone.

At the same time, all other modules conform to the following constraints: (a) they can perform non-transactional updates; (b) they ignore the rows with the *tombstone* marker to avoid updating the rows that are slated for deletion; (c) a single iteration of a module cannot remain live for more than 24 hours (our module scheduling mechanism enforces this).

This design satisfies the three conditions above while preserving the efficiency of the entire system.

5. FRONT END: SERVING THE CATALOG

We have so far focused on the process for building and maintaining the Goods catalog. In this section, we describe the main services enabled by the metadata that we collect in Goods (cf. the top part of Figure 1).

5.1 Dataset profile pages

The first service is to export the metadata for a specific dataset in an easy-to-view *profile page* for the dataset. Specifically, the profile-page service accepts as input the path of a dataset or a dataset cluster and generates an HTML page from the metadata stored in the catalog. The service also provides methods to edit specific parts of the metadata, to allow users either to augment or to correct the information stored in the catalog.

The profile page renders most of the dataset metadata that we described in Section 3. When presenting the profile page to the users, we must balance the need to present the metadata in a comprehensive way with the desire to keep the amount of information on the page manageable. The manageable size is important both not to overwhelm the users with too much information and to avoid transferring large amounts of information. Consider provenance information, for example: Popular datasets may be read by tens of thousands of jobs and have tens of thousands of datasets downstream from them. Incidentally, jobs such as Goods modules, which need to access every dataset in the company, have potentially billions of datasets upstream from them. To avoid transferring and trying to present such overwhelming amounts of information to the users (which would be futile to do anyway), we compress provenance metadata offline, whenever it becomes too large, using the same abstraction mechanism that we introduced in Section 3.2. We then use this compressed provenance to render the profile page. If the compressed version is still too large, then our last resort is to retain only a number of most recent entries.

The profile page of a dataset cross-links some of the metadata with other, more specialized, tools. For example, the profile page links the provenance metadata, such as jobs that generated the dataset, to the pages with details for those jobs in job-centric tools. Similarly, we link schema metadata to code-management tools, which provide the definition of this schema. Correspondingly, these tools link back to Goods to help users get more information about datasets.

The profile page also provides *access snippets* in different languages (e.g., C++, Java, SQL) to access the contents of the dataset. We custom-tailor the generated snippets for the specific dataset: For example, the snippets use the path and schema of the dataset (when known), and users can copy-paste the snippets in their re-

spective programming environment. The goal behind these snippets is to complement the content metadata in the profile page: the latter provide schema-level information about the contents of the dataset, whereas the snippets provide a handy way to inspect quickly the actual contents or to analyze the contents through code.

Overall the intent of a profile page is to provide a one-stop shop where a user can inspect the information about a dataset and understand the context in which the dataset can be used in production. This feature makes the profile page a natural handle for sharing a dataset among users or linking to dataset information from other tools. As an example of the latter, Google's filesystem browser provides direct links to Goods dataset-profile pages when users examine the contents of a directory.

5.2 Dataset search

Profile pages allow users to view information about specific datasets, but how can a user find datasets of interest? This task is where our dataset-search service comes in.

Dataset search allows Googlers to find datasets using simple keyword queries. The service is backed by a conventional inverted index for document retrieval, where each dataset becomes a "document" and we derive the indexing tokens for each document from a subset of the dataset's metadata. As is common in this context, each token can be associated with a specific section of the index. For example, a token derived from the path of the dataset is associated with the "path" section of the index. Accordingly, the search atom "path:x" will match keyword "x" on dataset paths only, whereas the unqualified atom "x" will match the keyword in any part of a dataset's metadata. Table 4 summarizes the main sections in the dataset-search index and their meaning in queries.

The extraction of indexing tokens follows from the type of queries that the index must cover. As an example, we want to support partial matches on the dataset path, where a user may search for "x/y" to match a dataset with path "a/x/y/b" (but not one with "a/y/x/b"). One approach is to index every sub-sequence of the path along common separators (e.g., for path "a/x/y/b" extract the indexing tokens "a/x","x/y", ..., "a/x/y", "x/y/b", and so on). However, this approach results in a blow-up in index size. Instead, we break up the dataset's path along common separators and then associate each resulting token with its position in the path. Going back to our example, the path "a/x/y/b" gets mapped to the indexing tokens "a", "x", "y", and "b", in that sequence. When the user issues a search query with a partial path, our service parses the partial path the same way and matches the query's tokens against consecutive tokens in the index. We followed a similar scheme when indexing the names of protocol buffers, which can be qualified with a namespace (e.g., "foo.bar.X"). In this fashion, a user can search for all datasets whose schema matches any protocol buffer under a specific namespace.

Matching search keywords to datasets is only the first part of the search task. The second part is deriving a scoring function to rank the matching datasets, so that the top results are relevant for the user's search. Scoring is generally a hard problem and part of our ongoing work involves tuning the scoring function based on our users' experience. In what follows, we describe some heuristics that informed the design of the scoring function thus far.

- *The importance of a dataset depends on its type.* For instance, our scoring function favors a Dremel table [19] over a file dataset, all else being equal. The intuition is that a dataset owner has to register a dataset as a Dremel table explicitly, which in turn makes the dataset visible to more users. We interpret this action as a signal that the dataset is important and reflect it in our final scoring.

Qualified token	Where token matches
path:*a*	Path of the dataset
proto:*a*	Name of protocol buffer
read_by:*a* written_by:*a*	Names of jobs reading/writing the dataset
downstream_of:*a* upstream_of:*a*	Paths of datasets downstream/upstream of the dataset
kind:*a*	Type of the dataset
owner:*a*	Owners of the dataset

Table 4: Examples of qualifying a search token *a* so that it matches different sections of the index. A search query may comprise several qualified and unqualified tokens.

- *The importance of a keyword match depends on the index section.* For instance, a keyword match on the path of the dataset is more important than a match on jobs that read or write the dataset, all else being equal. This heuristic reflects the types of searches that we observe in practice.

- *Lineage fan-out is a good indicator of dataset importance.* Specifically, this heuristic favors datasets with many reading jobs and many downstream datasets. The intuition is that if many production pipelines access the dataset, then most likely the dataset is important. One can view this heuristic as an approximation of PageRank in a graph where datasets and production jobs are vertices and edges denote dataset accesses from jobs. An interesting artifact of this heuristic and the nature of Google's production pipelines (and potentially pipelines in other enterprises as well) is assignment of inordinately high scores to certain datasets simply because they are consumed indirectly by many internal pipelines—even if they may not be useful to most users. One such example is Google's Web crawl, which numerous pipelines consume (often indirectly) in order to extract different types of information. Hence, it becomes important to tune and control the contribution of this heuristic to the overall ranking function, otherwise these datasets tend to get ranked high even for vaguely related searches.

- *A dataset that carries an owner-sourced description is likely to be important.* Our user interface enables dataset owners to provide descriptions for datasets that they want other teams to consume. We treat the presence of such a description as a signal of dataset importance. If a keyword match occurs in the description of a dataset then this dataset should be weighted higher as well.

Our scoring function incorporates these heuristics as well as other signals, and as with any similar setting, tuning the contribution of different signals has been an ongoing effort in our team. Note that we do not claim that the heuristics that we mentioned are complete. In fact, an interesting research problem is understanding the factors that influence dataset importance at search time (when the search keywords provide some indication of the user's intent) but also in a static, more global context that considers the usage of a dataset in relation to other datasets, jobs, and teams in the enterprise. This static context is relevant for our backend, as an additional signal on what datasets we should prioritize for metadata extraction.

In addition to the keyword search, Goods presents metadata facets for some of the categorical metadata in the catalog, such as dataset owners and dataset file formats. These facets give users an overview of the matching datasets, and help them formulate subsequent drill down queries significantly easier.

5.3 Team dashboards

The Goods dashboard is a configurable one-stop shop for displaying all the datasets generated by a team along with interesting metadata per dataset, such as various health metrics, other dashboards, and whether or not the storage system in which the dataset resides is online. Goods updates the content of a dashboard automatically as it updates the metadata of the datasets in the dashboard. Users can easily embed the dashboard page within other documents and share the dashboard with others.

The Goods dashboard also provides the means to monitor datasets and fire alerts if certain expected properties fail to hold (e.g., a dataset should have a certain number of shards or should have a certain distribution of values). Users can set up this type of monitoring with a few clicks and Goods is then responsible for checking the monitored properties for the corresponding datasets and to propagate any alerts to an in-house monitoring UI. In addition to performing fixed validation checks, Goods can generate checks by learning trends for a few common properties of interest. For example, if the size of a dataset historically increases by 10% for each version, then Goods can recommend a corresponding check that the next dataset size should be within some range around the projected size.

6. LESSONS LEARNED

In our efforts to build Goods we encountered many pitfalls—some that were avoidable and some that were hard to anticipate. In this section, we summarize some of the lessons that we have learned along the way.

Evolve as you go — We started building the catalog with dataset discovery as the target use case. Soon we realized that engineers were using Goods in a variety of ways, some of which deviated from or refined the initial use case. Here are the main trends that we observed in our usage logs or learned from our users:

- *Audit protocol buffers* Certain protocol buffers may contain personally identifiable information and so any datasets using these protocol buffers must adhere to strict usage and access policies. Using Goods, engineers can easily find all datasets conforming to a sensitive protocol buffer and alert the dataset owners in case of policy violations.

- *Re-find datasets* Engineers generate many "experimental" datasets as part of their work, but often forget the paths when they want to share these datasets or continue working on them. Usually these datasets can be easily re-found with a simple keyword search.

- *Understand legacy code* Up-to-date documentation for legacy code can be hard to find. Goods exposes a provenance graph that engineers can use to track previous executions of legacy code along with input and output datasets, which in turn can provide useful clues for the underlying logic.

- *Bookmark datasets* A dataset's profile page is a natural one-stop-shop for information about the dataset. Users bookmark these pages for easy access and also to share datasets with other users.

- *Annotate datasets* The Goods catalog serves as a hub for dataset annotations that can be shared across teams. For example, teams can tag their datasets with different levels of privacy in order to warn engineers about the intended usage of the datasets and also to facilitate policy checks.

It is noteworthy that the last feature was built by a different team within Google, one who piggybacked on the infrastructure that we developed. This outside contribution validated our goal to help create a company-wide ecosystem of tools around dataset management.

As we were developing Goods, we had several meetings with teams inside Google to discuss their pain points in dataset management. We realized very quickly the need for a holistic suite of data-management tools beyond search, including dashboards to monitor dataset health, automated dataset testing, and tools to understand differences among datasets. Our extensible design enabled us to support some of these use cases with a relatively small incremental effort. Moreover, some of these tools can enhance the catalog with additional metadata that apply in other use cases. For example, datasets that users explicitly included in data dashboards for monitoring were clearly important datasets and therefore we could boost their ranking.

Use domain-specific signals for ranking — As mentioned in Section 2, the problem of ranking datasets has unique characteristics when compared to ranking problems in other domains (e.g., Web ranking). Our experience with Goods confirmed this observation. For example, we have found that provenance relationships among datasets provide a strong domain-specific ranking signal. Specifically, it is common for teams to generate denormalized versions of some "master" dataset in order to facilitate different types of analysis of the master data. These denormalized datasets can match the same search keywords as the master dataset, yet it is clear that the master dataset should be ranked higher for general-purpose queries or for metadata-extraction. Another example comes from provenance relationships that cross team boundaries, when the dataset from one team is processed to create a dataset in the scope of another team or project. In this case, we can boost the importance of the input dataset as evidenced by its usage by an external team. The output dataset is also important, since we can view it as an origin of other datasets within the external project.

Identifying the type of provenance relationships is an interesting research problem, especially in the context of post-hoc metadata inference. There are several signals that we can leverage in this direction, including content similarity among datasets, known provenance relationships, and crowdsourced information such as the owner-provided descriptions. Once we categorize these relationships we then have to reason about them in the context of ranking, a problem that itself involves a different set of challenges.

Expect and handle unusual datasets — Given the large number of datasets in the catalog, we ran into many unexpected scenarios in our early days. We solved some of them as one-off cases with dedicated code while others required a system-level redesign. For example, the abstraction mechanism described in Section 3.2 employs special logic to extract out unconventionally specified dates (for example, "05Jan2015") and versions. Handling of certain datasets that caused our analyzers to crash due to issues in third party libraries outside our codebase required the sandboxing mechanism described in Section 4.3. We adopted the strategy of going with the simplest albeit adhoc solution first and generalized it as and when required.

Export data as required — The storage medium for the Goods catalog is a key-value store and the search service is backed by a conventional inverted index. However, neither of these structures are suitable for visualizing or performing complex path queries over the provenance graph. To support such use cases, we set up a daily export of the catalog data as subject–predicate–object triples. We then import these triples into a graph-based system that supports path queries and exposes an API that is more amenable for

visualization. For use cases that require more powerful query processing capabilities that our storage medium does not support natively, the easiest strategy is to export the catalog data to a suitably specialized engine.

Ensure recoverability — Extracting the metadata of billions of datasets is expensive. In stable state, we process one day's worth of new datasets in a single day. Losing or corrupting a significant chunk of the catalog can take weeks to recover unless we dedicate significant additional computational resources to recovery. Moreover, we may not even be able to recompute some of the metadata after significant data loss. For example, some transient files link input and output datasets in the provenance graph. If we loose the provenance data that we inferred from these transient files, we will not be able to recover it.

To ensure recoverability we have configured Bigtable to retain a rolling window of snapshots over several days, but we have also built custom-tailored recovery schemes specifically for Goods. Specifically, we have added a dedicated process to snapshot high-value datasets (those in which users have explicitly expressed interest through annotations) in a separate catalog in order to guard against data loss. Another process replicates the subset of the catalog that powers the profile pages so that the service remains available even if the main catalog goes offline. Furthermore, we use the Goods dataset-monitoring service for the catalog itself (which is another structured dataset!) to ensure early detection of data corruption and deletion. We arrived at this combined approach based on our experience of repairing the catalog on several occasions, some of which caused major outages to the user-facing services.

7. RELATED WORK

We can characterize Goods as a data lake, an approach to store massive amounts of data in easy-to-access repositories, without having to pre-categorize them when they are created. Specifically, Goods is a system to organize and index the data lake of all Google datasets. Similar efforts are underway in other companies, such as the data-lake management system developed in IBM Research [22] or data-lake offerings in cloud services [1]. We do not have the complete details for these systems, but the distinguishing features of Goods seem to be the scale of the lake and the post-hoc approach to metadata inference. Goods also shares the goals and ideas of Dataspaces [16], and we can view it as a concrete implementation of that idea.

DataHub [8, 9, 10] is a collaborative version-management system for datasets that is similar in spirit to software version control systems like Git and SVN. DataHub enables many users to analyze, collaborate, modify, and share datasets in a centralized repository. Data-management systems such as CKAN [3], Quandl [5], and Microsoft Azure Marketplace [2] are repositories of data from multiple sources, organized for distribution and sharing. In all these systems, dataset owners actively choose to contribute their datasets to the system or to annotate the datasets with metadata. The main focus of Goods is to create a catalog in a post-hoc manner, while the engineers continue to create and maintain the datasets in their current environment. The engineers do not need to change any of their practices in order for Goods to provide access to their datasets.

Researchers have proposed several systems [12, 26, 7] to extract HTML tables from the Web, to annotate their schema, and to implement a search service for this type of Webtables. However, much of the machinery in Webtable systems is devoted to the identification of HTML tables that hold data (and are not merely used for layouts), something that is not necessary for the kind of structured datasets that Goods handles. Furthermore, metadata such as provenance, owners, or project affiliation are absent for Webtables but are very important for datasets.

Fast and efficient file search has become necessary for storage systems. Spyglass [18] is a metadata-search system that improves file management by allowing complex, ad hoc queries over file metadata. Propeller [25] also provides fast file-search services and updates its file indices in a real-time and scalable fashion. In comparison, the datasets in Goods are not in a single storage system and therefore do not have as much readily available metadata, making search a significantly more challenging problem. In addition, Goods aims to provide a suite of tools for dataset management that go beyond search.

Previous studies have looked into the problem of indexing and searching a large corpus of structured datasets [21, 17]. The developed techniques could be used to power the search service of Goods, but they would not solve the main challenges that we outlined in Section 2. However, these techniques may become more relevant if Goods needs to index the actual content of datasets, in which case the catalog will increase significantly in size.

Provenance management has been studied extensively in the literature [14]. Some examples are the PASS [20] and Trio [24] systems to maintain the provenance of files and database records, respectively, and Datahub [10] as a system to maintain versions of datasets. These systems assume that provenance information is either given or can be retrieved reliably through special instrumentation of the processes that access the data. By contrast, Goods has to rely on much weaker signals for provenance inference while also relying on provenance as a strong signal for dataset ranking.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have described a system that provides access to metadata about billions of datasets within an enterprise. As might be expected with a system of this scale, we had to solve many challenges—some that we expected and some that we had not. The resulting system provides real value for engineers at Google and already supports many data-management use cases—again, some that we had anticipated and some that grew out of feedback from users who were using the system.

Many significant challenges remain, however. First, we are still far from fully understanding how to rank datasets, how to identify important datasets, and whether or not approaches are similar to ranking of web pages. We discussed some of the signals that we use for ranking in Section 5.2, but we know from the users' feedback that we must improve ranking significantly. We need to be able to distinguish between production and test or development datasets, between datasets that provide input for many other datasets, datasets that users care about, and so on.

Second, because the teams generate their datasets independently and some provide a lot more metadata about the data than others, we need to rely on other, weaker, signals in order to fill the missing metadata about a dataset. We plan to integrate the data with a larger knowledge graph that links together many of the Google's resources, such as datasets, code, and teams. For instance, knowing which code generated a dataset can provide clues to its content. Knowing what are the projects that the team that is responsible for a dataset works on can fill out some of the missing metadata. We believe that there are multiple connections like these ones that would allow us to propagate some of the missing metadata between related entities.

Third, understanding the semantics of the data inside the datasets is another source of information that would make our services—search, in particular—more useful to the users. Again, we can rely on other knowledge structures at Google, such as the Knowledge

Graph (which other companies also have) to gain some understanding of these semantics.

Fourth, modifying existing storage systems to register their datasets with Goods at the time when the datasets are created can improve the freshness of the catalog. This approach, however, is contrary to the non-invasive post-hoc approach that we have described in this paper. Exploring a combination of the two approaches is future work.

Finally, we hope that systems such as Goods will provide an impetus to instilling a "data culture" at data-driven companies today in general, and at Google in particular. As we develop systems that enable enterprises to treat datasets as core assets that they are, through dashboards, monitoring, and so on, it will hopefully become as natural to have as much "data discipline" as we have "code discipline."

9. ACKNOWLEDGMENTS

We would like to thank John Wilkes for his insightful comments which helped us improve the paper. We would also like to thank Lorenzo Martignoni and Andi Vajda who helped us in collecting and visualizing provenance metadata, respectively. Finally, we would like to thank our interns, Dominik Moritz, Arun Iyer, Xiao Cheng, Hui Miao, Subhabrata Mukherjee and Jaeho Shin, who have contributed to the features of our system.

10. REFERENCES

[1] Azure data lake. https://azure.microsoft.com/en-us/solutions/data-lake/.

[2] Azure marketplace. http://datamarket.azure.com/browse/data.

[3] CKAN. http://ckan.org.

[4] Data lakes and the promise of unsiloed data. http://www.pwc.com/us/en/technology-forecast/2014/cloud-computing/features/data-lakes.html.

[5] Quandl. https://www.quandl.com.

[6] A universally unique identifier (uuid) urn namespace. https://www.ietf.org/rfc/rfc4122.txt.

[7] S. Balakrishnan, A. Y. Halevy, B. Harb, H. Lee, J. Madhavan, A. Rostamizadeh, W. Shen, K. Wilder, F. Wu, and C. Yu. Applying webtables in practice. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA*, 2015.

[8] A. P. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. DataHub: Collaborative data science & dataset version management at scale. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA*, 2015.

[9] A. P. Bhardwaj, A. Deshpande, A. J. Elmore, D. R. Karger, S. Madden, A. G. Parameswaran, H. Subramanyam, E. Wu, and R. Zhang. Collaborative data analytics with DataHub. *PVLDB*, 8(12):1916–1927, 2015.

[10] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. G. Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *PVLDB*, 8(12):1346–1357, 2015.

[11] A. Brown. Get smarter answers from the knowledge graph. http://insidesearch.blogspot.com/2012/12/get-smarter-answers-from-knowledge_4.html, 2012.

[12] M. J. Cafarella, A. Y. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. Webtables: exploring the power of tables on the web. *PVLDB*, 1(1):538–549, 2008.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[14] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Found. Trends databases*, 1(4):379–474, Apr. 2009.

[15] P. Flajolet, E. Fusy, G. O., and F. Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. *Analysis of Algorithms (AOFA)*, 2007.

[16] M. Franklin, A. Halevy, and D. Maier. From databases to dataspaces: A new abstraction for information management. *SIGMOD Rec.*, 34(4):27–33, Dec. 2005.

[17] I. Konstantinou, E. Angelou, D. Tsoumakos, and N. Koziris. Distributed indexing of web scale datasets for the cloud. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, MDAC '10, pages 1:1–1:6, 2010.

[18] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. In M. I. Seltzer and R. Wheeler, editors, *FAST*, pages 153–166. USENIX, 2009.

[19] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, 2011.

[20] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, pages 43–56, 2006.

[21] P. Rao and B. Moon. An internet-scale service for publishing and locating xml documents. In *Proceedings of the 2009 Int'l Conference on Data Engineering (ICDE)*, pages 1459–1462, 2009.

[22] I. Terrizzano, P. M. Schwarz, M. Roth, and J. E. Colino. Data wrangling: The challenging journey from the wild to the lake. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA*, 2015.

[23] K. Varda. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Accessed July*, 2008.

[24] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.

[25] L. Xu, H. Jiang, X. Liu, L. Tian, Y. Hua, and J. Hu. Propeller: A scalable metadata organization for a versatile searchable file system. Technical Report 119, Department of Computer Science and Engineering, University of Nebraska-Lincoln, 2011.

[26] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. InfoGather: entity augmentation and attribute discovery by holistic matching with web tables. In K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *SIGMOD Conference*, pages 97–108. ACM, 2012.