Ground: A Data Context Service

Joseph M. Hellerstein[*°], Vikram Sreekanti[*], Joseph E. Gonzalez[*], James Dalton[△], Akon Dey[♯], Sreyashi Nag[§], Krishna Ramachandran[♮], Sudhanshu Arora[‡], Arka Bhattacharyya[*], Shirshanka Das[†], Mark Donsky[‡], Gabe Fierro[*], Chang She[‡], Carl Steinbach[†], Venkat Subramanian[♭], Eric Sun[†]

[*]UC Berkeley, [°]Trifacta, [△]Capital One, [♯]Awake Networks, [§]University of Delhi, [♮]Skyhigh Networks, [‡]Cloudera, [†]LinkedIn, [♭]Dataguise

ABSTRACT

Ground is an open-source *data context service*, a system to manage all the information that informs the use of data. Data usage has changed both philosophically and practically in the last decade, creating an opportunity for new data context services to foster further innovation. In this paper we frame the challenges of managing data context with basic ABCs: *Applications*, *Behavior*, and *Change*. We provide motivation and design guidelines, present our initial design of a common metamodel and API, and explore the current state of the storage solutions that could serve the needs of a data context service. Along the way we highlight opportunities for new research and engineering solutions.

1. FROM CRISIS TO OPPORTUNITY

Traditional database management systems were developed in an era of risk-averse design. The technology itself was expensive, as was the on-site cost of managing it. Expertise was scarce and concentrated in a handful of computing and consulting firms.

Two conservative design patterns emerged that lasted many decades. First, the accepted best practices for deploying databases revolved around tight control of schemas and data ingest in support of general-purpose accounting and compliance use cases. Typical advice from data warehousing leaders held that *"There is no point in bringing data . . . into the data warehouse environment without integrating it"* [17]. Second, the data management systems designed for these users were often built by a single vendor and deployed as a monolithic stack. A traditional DBMS included a consistent storage engine, a dataflow engine, a language compiler and optimizer, a runtime scheduler, a metadata catalog, and facilities for data ingest and queueing—all designed to work closely together.

As computing and data have become orders of magnitude more efficient, changes have emerged for both of these patterns. Usage is changing profoundly, as expertise and control shifts from the central accountancy of an IT department to the domain expertise of "business units" tasked with extracting value from data [14]. The changes in economics and usage brought on the "three Vs" of Big Data: Volume, Velocity and Variety. Resulting best practices focus on open-ended schema-on-use data "lakes" and agile development, in support of exploratory analytics and innovative application intelligence [28]. Second, while many pieces of systems software that have emerged in this space are familiar, the overriding architecture is profoundly different. In today's leading open source data management stacks, nearly all of the components of a traditional DBMS are explicitly independent and interchangeable. This architectural decoupling is a critical and under-appreciated aspect of the Big Data movement, enabling more rapid innovation and specialization.

CIDR '17 January 8-11, 2017, Chaminade, CA, USA

1.1 Crisis: Big Metadata

An unfortunate consequence of the disaggregated nature of contemporary data systems is the lack of a standard mechanism to assemble a collective understanding of the origin, scope, and usage of the data they manage. In the absence of a better solution to this pressing need, the Hive Metastore is sometimes used, but it only serves simple relational schemas—a dead end for representing a Variety of data. As a result, data lake projects typically lack even the most rudimentary information about the data they contain or how it is being used. For emerging Big Data customers and vendors, this *Big Metadata* problem is hitting a crisis point.

Two significant classes of end-user problems follow directly from the absence of shared metadata services. The first is poor productivity. Analysts are often unable to discover what data exists, much less how it has been previously used by peers. Valuable data is left unused and human effort is routinely duplicated—particularly in a schema-on-use world with raw data that requires preparation. "Tribal knowledge" is a common description for how organizations manage this productivity problem. This is clearly not a systematic solution, and scales very poorly as organizations grow.

The second problem stemming from the absence of a system to track metadata is governance risk. Data management necessarily entails tracking or controlling who accesses data, what they do with it, where they put it, and how it gets consumed downstream. In the absence of a standard place to store metadata and answer these questions, it is impossible to enforce policies and/or audit behavior. As a result, many administrators marginalize their Big Data stack as a playpen for non-critical data, and thereby inhibit both the adoption and the potential of new technologies.

In our experiences deploying and managing systems in production, we have seen the need for a common service layer to support the capture, publishing and sharing of metadata information in a flexible way. The effort in this paper began by addressing that need.

1.2 Opportunity: Data Context

The lack of metadata services in the Big Data stack can be viewed as an opportunity: a clean slate to rethink how we track and leverage modern usage of data. Storage economics and schema-on-use agility suggest that the Data Lake movement could go much farther than Data Warehousing in enabling diverse, widely-used central

repositories of data that can adapt to new data formats and rapidly changing organizations. In that spirit, we advocate rethinking traditional metadata in a far more comprehensive sense. More generally, what we should strive to capture is the full context of data.

To emphasize the conceptual shifts of this *data context*, and as a complement to the "three Vs" of Big Data, we introduce three key sources of information—the *ABCs of Data Context*. Each represents a major change from the simple metadata of traditional enterprise data management.

Applications: Application context is the core information that describes how raw bits get interpreted for use. In modern agile scenarios, application context is often relativistic (many schemas for the same data) and complex (with custom code for data interpretation). Application context ranges from basic data descriptions (encodings, schemas, ontologies, tags), to statistical models and parameters, to user annotations. All of the artifacts involved—wrangling scripts, view definitions, model parameters, training sets, etc.—are critical aspects of application context.

Behavior: This is information about how data was created and used over time. In decoupled systems, behavioral context spans multiple services, applications and formats and often originates from high-volume sources (e.g., machine-generated usage logs). Not only must we track upstream lineage— the data sets and code that led to the creation of a data object—we must also track the downstream lineage, including data products derived from this data object. Aside from data lineage, behavioral context includes logs of usage: the "digital exhaust" left behind by computations on the data. As a result, behavioral context metadata can often be larger than the data itself.

Change: This is information about the version history of data, code and associated information, including changes over time to both structure and content. Traditional metadata focused on the present, but historical context is increasingly useful in agile organizations. This context can be a linear sequence of versions, or it can encompass branching and concurrent evolution, along with interactions between co-evolving versions. By tracking the version history of all objects spanning code, data, and entire analytics pipelines, we can simplify debugging and enable auditing and counterfactual analysis.

Data context services represent an opportunity for database technology innovation, and an urgent requirement for the field. We are building an open-source data context service we call *Ground*, to serve as a central model, API and repository for capturing the broad context in which data gets used. Our goal is to address practical problems for the Big Data community in the short term and to open up opportunities for long-term research and innovation.

In the remainder of the paper we illustrate the opportunities in this space, design requirements for solutions, and our initial efforts to tackle these challenges in open source.

2. DIVERSE USE CASES

To illustrate the potential of the Ground data context service, we describe two concrete scenarios in which Ground can aid in data discovery, facilitate better collaboration, protect confidentiality, help diagnose problems, and ultimately enable new value to be captured from existing data. After presenting these scenarios, we explore the design requirements for a data context service.

2.1 Scenario: Context-Enabled Analytics

This scenario represents the kind of usage we see in relatively technical organizations making aggressive use of data for machine-learning driven applications like customer targeting. In these organizations, data analysts make extensive use of flexible tools for data preparation and visualization and often have some SQL skills, while data scientists actively prototype and develop custom software for machine learning applications.

Janet is an analyst in the Customer Satisfaction department at a large bank. She suspects that the social network behavior of customers can predict if they are likely to close their accounts (customer churn). Janet has access to a rich *context-service-enabled* data lake and a wide range of tools that she can use to assess her hypothesis.

Janet begins by downloading a free sample of a social media feed. She uses an advanced data catalog application (we'll call it "Catly") which connects to Ground, recognizes the content of her sample, and notifies her that the bank's data lake has a complete feed from the previous month. She then begins using Catly to search the lake for data on customer retention: what is available, and who has access to it? As Janet explores candidate schemas and data samples, Catly retrieves usage data from Ground and notifies her that Sue, from the data-science team, had previously used a database table called `cust_roster` as input to a Python library called `cust_churn`. Examining a sample from `cust_roster` and knowing of Sue's domain expertise, Janet decides to work with that table in her own churn analysis.

Having collected the necessary data, Janet turns to a data preparation application ("Preply") to clean and transform the data. The social media data is a JSON document; Preply searches Ground for relevant wrangling scripts and suggests unnesting attributes and pivoting them into tables. Based on security information in Ground, Preply warns Janet that certain customer attributes in her table are protected and may not be used for customer retention analysis. Finally, to join the social media names against the customer names, Preply uses previous wrangling scripts registered with Ground by other analysts to extract standardized keys and suggest join conditions to Janet.

Having prepared the data, Janet loads it into her BI charting tool and discovers a strong correlation between customer churn and social sentiment. Janet uses the "share" feature of the BI tool to send it to Sue; the tool records the share in Ground.

Sue has been working on a machine learning pipeline for automated discount targeting. Janet's chart has useful features, so Sue consults Ground to find the input data. Sue joins Janet's dataset into her existing training data but discovers that her pipeline's prediction accuracy *decreases*. Examining Ground's schema for Janet's dataset, Sue realizes that the `sentiment` column is categorical and needs to be pivoted into indicator columns `isPositive`, `isNegative`, and `isNeutral`. Sue writes a Python script to transform Janet's data into a new file in the required format. She trains a new version of the targeting model and deploys it to send discount offers to customers at risk of leaving. Sue registers her training pipeline including Janet's social media feeds in the daily build; Ground is informed of the new code versions and service registration.

After several weeks of improved predictions, Sue receives an alert from Ground about changes in Janet's script; she also sees a notable drop in prediction accuracy of her pipeline. Sue discovers that some of the new social media messages are missing sentiment scores. She queries Ground for the version of the data and pipeline code when sentiment scores first went missing. Upon examination, she sees that the upgrade to the sentiment analysis code produced new categories for which she doesn't have columns (e.g., `isAngry`, `isSad`, ...). Sue uses Ground to roll back the sentiment analysis code in Janet's pipeline and re-run her pipeline for the past month. This fixes Sue's problem, but Sue wonders if she can simply roll back Janet's scripts in production. Consulting Ground, Sue discovers that other pipelines now depend upon the new version of Janet's scripts. Sue calls a meeting with the relevant stakeholders to untangle the situation.

Throughout our scenario, the users and their applications benefited from global data context. Applications like Catly and Preply were able to provide innovative features by mining the "tribal knowledge" captured in Ground: recommending datasets and code, identifying experts, flagging security concerns, notifying developers of changes, etc. The users were provided contextual awareness of both technical and organizational issues and able to interrogate global context to understand root causes. Many of these features exist in isolated applications today, but would work far better with global context. Data context services make this possible, opening up opportunities for innovation, efficiency and better governance.

2.2 Scenario: Big Data in Enterprise IT

Many organizations are not as technical as the one in our previous scenario. We received feedback on an early draft of this paper from an IT executive at a global financial services firm (not affiliated with the authors), who characterized both Janet and Sue as "developers" not analysts. ("If she knows what JSON is, she's a developer!") In his organization, such developers represent less than 10% of the data users. The remaining 90% interact solely with graphical interfaces. However, he sees data context offering enormous benefits to his organization. Here we present an illustrative enterprise IT scenario.

Mark is an Data Governance manager working in the IT department of a global bank. He is responsible for a central data warehouse, and the legacy systems that support it, including Extract-Transform-Load (ETL) mappings for loading operational databases into the warehouse, and Master Data Management (MDM) systems for governing the "golden master" of various reference data sets (customers, partner organizations, and so on.) Recently, the bank decided to migrate off of these systems and onto a Big Data stack, to accomodate larger data volumes and greater variety of data. In so doing, they rewrote many of their workflows; the new workflows register their context in Ground.

Sara is an analyst in the bank's European Compliance office; she uses Preply to prepare monthly reports for various national governments demonstrating the firm's compliance with regulations like Basel III [35]. As Sara runs this month's `AssetAllocation` report, she sees that a field called `IPRE_AUSNZ` came back with a very small value relative to other fields prefixed with `IPRE`. She submits a request to the IT department's trouble ticket system ("Helply") referencing the report she ran, asking "What is this field? What are the standard values? If it is unusual, can you help me understand why?" Mark receives the ticket in his email, and Helply stores an association in Ground between Sara and `AssetAllocation`. Mark looks in Ground at summary statistics for the report fields over time, and confirms that the value in that field is historically low by an order of magnitude. Mark then looks at a "data dictionary" of reference data in Ground and sees that `IPRE` was documented as "Income-Producing Real Estate". He looks at lineage data in Ground and finds that the `IPRE_AUSNZ` field in the report is calculated by a SQL view aggregating data from both Australia and New Zealand. He also looks at version information for the view behind `AssetAllocation`, and finds that the view was modified on the second day of the month to compute two new fields, `IPRE_AUS` and `IPRE_NZ` that separate the reporting across those geographies. Mark submits a response in Helply that explains this to Sara. Armed with that information, Sara uses the Preply UI to sum all three fields into a single cell representing the IPRE calculation for the pair of countries over the course of the full month.

Based on the Helply association, Sara is subscribed automatically to an RSS feed associated with `AssetAllocation`. In future, Sara will automatically learn about changes that affect the report, thanks to the the new workloads from Mark's team that auto-generate data lineage in Ground. Mark's team takes responsibility for *upstream* reporting of version changes to data sources (e.g. reference data) and code (ETL scripts, warehouse queries, etc), as well as the data lineage implicit in that code. Using that data lineage, a script written by Mark's team auto-computes *downstream* Helply alerts for all data products that depend transitively on a change to upstream data and scripts.

In this scenario, both the IT and business users benefit from various kinds of context stored in Ground, including statistical data profiles, data dictionaries, field-level data lineage, code version history, and (transitive) associations between people, data, code and their versions. Our previous data science use cases largely exploited statistical and probabilistic aspects of context (correlations, recommendations); in this scenario, the initial motivation was quantitative, but the context was largely used in more deterministic and discrete ways (dependencies, definitions, alerts). Over time time, we believe organizations will leverage data context using both deterministic and probabilistic approaches.

3. DESIGN AND ARCHITECTURE

In a decoupled architecture of multiple applications and backend services, context serves as a "narrow waist"—a single point of access for the basic information about data and its usage. It is hard to anticipate the breadth of applications that could emerge. Hence we were keen in designing Ground to focus on initial decisions that could enable new services and applications in future.

3.1 Design Requirements

In our design, we were guided by Postel's Law of Robustness from Internet architecture: *"Be conservative in what you do, be liberal in what you accept from others."* Guided by this philosophy, we identified four central design requirements for a successful data context service.

Model-Agnostic. For a data context service to be broadly adopted, it cannot impose opinions on metadata modeling. Data models evolve and persist over time: modern organizations have to manage everything from COBOL data layouts to RDBMS dumps to XML, JSON, Apache logs and free text. As a result, the context service cannot prescribe how metadata is modeled—each dataset may have different metadata to manage. This is a challenge in legacy "master data" systems, and a weakness in the Big Data stack today: Hive Metastore captures fixed features of relational schemas; HDFS captures fixed features of files. A key challenge in Ground is to design a core metamodel that captures generic information that applies to all data, as well as custom information for different data models, applications, and usage. We explore this issue in Section 3.3.

Immutable. Data context must be immutable; *updating* stored context is tantamount to erasing history. There are multiple reasons why history is critical. The latest context may not always be the most relevant: we may want to replay scenarios from the past for what-if analysis or debugging, or we may want to study how context information (e.g., success rate of a statistical model) changes over time. Prior context may also be important for governance and veracity purposes: we may be asked to audit historical behavior and metadata, or reproduce experimental results published in the past. This simplifies record-keeping, but of course it raises significant engineering challenges. We explore this issue in Section 4.

Scalable. It is a frequent misconception that metadata is small. In fact, metadata scaling was already a challenge in previous-generation ETL technology. In many Big Data settings, it is reasonable to envision the data context being far larger than the data

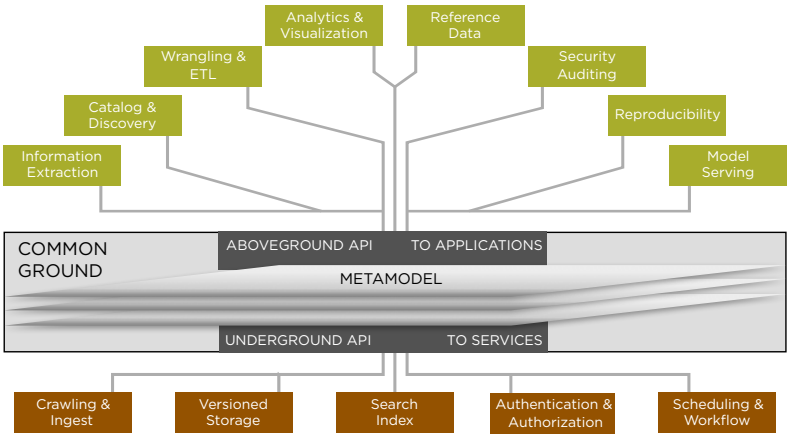

Figure 1: The architecture of Ground. The Common Ground metamodel (Section 3.3) is at the center, supported by a set of swappable underground services. The system is intended to support a growing set of aboveground applications, examples of which are shown. Ground is decoupled from applications and services via asynchronous messaging services. Our initial concrete instantiation of this architecture, Ground 0, is described in Section 4.

itself. Usage information is one culprit: logs from a service can often outstrip the data managed by the service. Another is data lineage, which can grow to be extremely large depending on the kind of lineage desired [8]. Version history can also be substantial. We explore these issues in Section 4 as well.

Politically Neutral. Common narrow-waist service like data context must interoperate with a wide range of other services and systems designed and marketed by often competing vendors. Customers will only adopt and support a central data context service if they feel no fear of lock-in; application writers will prioritize support for widely-used APIs to maximize the benefit of their efforts. It is important to note here that open source is not equivalent to political neutrality; customers and developers have to believe that the project leadership has strong incentives to behave in the common interest.

Based on the requirements above, the Ground architecture is informed by Postel's Law of Robustness and the design pattern of decoupled components. At its heart is a foundational metamodel called *Common Ground* with an associated *aboveground* API for data management applications like the catalog and wrangling examples above. The core functions underneath Ground are provided by swappable component services that plug in via the *underground* API. A sketch of the architecture of Ground is provided in Figure 1.

3.2 Key Services

Ground's functionality is backed by five decoupled subservices, connected via direct REST APIs and a message bus. For agility, we are starting the project using existing open source solutions for each service. We anticipate that some of these will require additional features for our purposes. In this section we discuss the role of each subservice, and highlight some of the research opportunities we foresee. Our initial choices for subservices are described in Section 4.

Ingest: Insertion, Crawlers and Queues. Metadata may be pushed into Ground or require crawling; it may arrive interactively via REST APIs or in batches via a message bus. A main design decision is to decouple the systems plumbing of ingest from an extensible set of metadata and feature extractors. To this end, ingest has both underground and aboveground APIs. New context metadata arrives for ingestion into Ground via an underground queue API from crawling services, or via an aboveground REST API from applications. As metadata arrives, Ground publishes notifications via an aboveground queue. aboveground applications can subscribe to these events to add unique value, fetching the associated metadata and data, and generating enhanced metadata asynchronously. For example, an application can subscribe for file crawl events, hand off the files to an entity extraction system like OpenCalais or AlchemyAPI, and subsequently tag the corresponding Common Ground metadata objects with the extracted entities.

Metadata feature extraction is an active research area; we hope that commodity APIs for scalable data crawling and ingest will drive more adoption and innovation in this area.

Versioned Metadata Storage. Ground must be able to efficiently store and retrieve metadata with the full richness of the Common Ground metamodel, including flexible version management of code and data, general-purpose model graphs and lineage storage. While none of the existing open source DBMSs target this data model, one can implement it in a shim layer above many of them. We discuss this at greater length in Section 4.1, where we examine a range of widely-used open source DBMSs. As noted in that section, we believe this is an area for significant database research.

Search and Query. Access to context information in Ground is expected to be complex and varied. As is noted later, Common Ground supports arbitrary tags, which leads to a requirement for search-style indexing that in current open source is best served by an indexing service outside the storage system. Second, intelligent applications like those in Section 2 will run significant analytical workloads over metadata—especially usage metadata which could be quite large. Third, the underlying graphs in the Common Ground model require support for basic graph queries like transitive closures. Finally, it seems natural that some workloads will need to combine these three classes of queries. As we explore in Section 4.1, various open-source solutions can address these workloads at some level, but there is significant opportunity for research here.

Authentication and Authorization. Identity management and authorization are required for a context service, and must accommodate typical packages like LDAP and Kerberos. Note that authorization needs vary widely: the policies of a scientific consortium will differ from a defense agency or a marketing department. Ground's flexible metamodel can support a variety of relevant metadata (ownership,

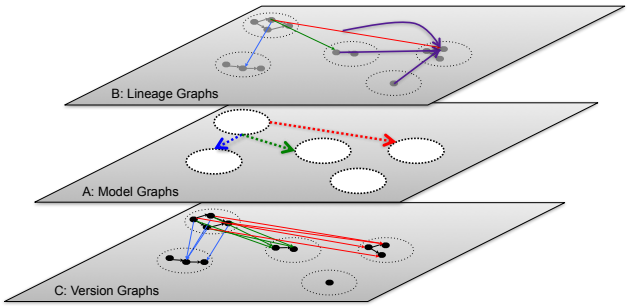

Figure 2: The Common Ground metamodel.

content labels, etc.) Meanwhile, the role of versioning raises subtle security questions. Suppose the authorization policies of a past time are considered unsafe today—should reproducibility and debugging be disallowed? More research is needed integrate versions and lineage with security techniques like Information Flow Control [26] in the context of evolving real-world pipelines.

Scheduling, Workflow, Reproducibility. We are committed to ensuring that Ground is flexible enough to capture the specification of workflows at many granularities of detail: from black-box containers to workflow graphs to source code. However, we do not expect Ground to be a universal provider of workflow execution or scheduling; instead we hope to integrate with a variety of schedulers and execution frameworks including on-premises and cloud-hosted approaches. This is currently under design, but the ability to work with multiple schedulers has become fairly common in the open source Big Data stack, so this may be a straightforward issue.

3.3 The Common Ground Metamodel

Ground is designed to manage both the ABCs of data context and the design requirements of data context services. The Common Ground metamodel is based on a layered graph structure shown in Figure 2: one layer for each of the ABCs of data context.

3.3.1 Version Graphs: Representing Change

We begin with the version graph layer of Common Ground, which captures changes corresponding to the *C* in the ABCs of data context (Figure 3). This layer bootstraps the representation of all information in Ground, by providing the classes upon which all other layers are based. These classes and their subclasses are among the only information in Common Ground that is not itself versioned; this is why it forms the base of the metamodel.

The main atom of our metamodel is the `Version`, which is simply a globally unique identifier; it represents an immutable version of some object. We depict `Version`s via the small circles in the bottom layer of Figure 2. Ground links `Version`s into `VersionHistoryDAG`s via `VersionSuccessor` edges indicating that one version is the descendant of another (the short dark edges in the bottom of Figure 2.) Type parametrization ensures that all of the `VersionSuccessor`s in a given DAG link the same subclass of `Version`s together. This representation of DAGs captures any partial order, and is general enough to reflect multiple different versioning systems.

`RichVersion`s support customization. These variants of `Version`s can be associated with ad hoc `Tag`s (key-value pairs) upon creation. Note that all of the classes introduced above are immutable—new values require the creation of new `Version`s.

External Items and Schrödinger Versioning
We often wish to track items whose metadata is managed outside

```
1  public class Version {
2      private String id;
3  }

5  public class VersionSuccessor<T extends Version> {
6      // the unique id of this VersionSuccessor
7      private String id;
8      // the id of the Version that originates this successor
9      private String fromId;
10     // the id of the Version that this success points to
11     private String toId;
12 }

14 public class VersionHistoryDAG<T extends Version> {
15     // the id of the Version that's at the rootId of this DAG
16     private String itemId;
17     // list of VersionSuccessors that make up this DAG
18     private List<String> edgeIds;
19     // map of parents to children
20     private Map<String, List<String>> parentChildMap;
21 }

23 public class RichVersion extends Version {
24     // the map of Keys to Tags associated with this RichVersion
25     private Map<String, Tag> tags;
26     // the StructureVersion associated with this RichVersion
27     private String structureVersionId;
28     // the optional reference associated with this RichVersion
29     private String reference;
30     // the optional parameters if this is an external version
31     private Map<String, String> parameters;
32     // timestamp if this is an external version
33     private Instant externalAccessTimeStamp;
34     // optional cache of the external version
35     private Object cachedValue;
36 }

38 public class Tag {
39     private String versionId;
40     // the Key of the Tag
41     private String key;
42     // the optional Value of the Tag
43     private Object value;
44     // the Type of the Value if it exists
45     private GroundType valueType;
46 }
```

Figure 3: Java skeleton for Version classes in Common Ground. Methods have been elided. Full code is available at `https://github.com/ground-context/ground`.

of Ground: canonical examples include GitHub repositories and Google Docs. Ground cannot automatically track these items as they change; at best it can track observations of those items. Observed versions of external items are represented by optional fields in Ground's `RichVersion`s: the `parameters` for accessing the reference (e.g., port, protocol, URI, etc.), an `externalAccessTimestamp`, and an optional `cachedValue`. Whenever a Ground client uses the aboveground API to access a `RichVersion` with non-empty external `parameters`, Ground fetches the external object and generates a new `ExternalVersion` containing a new `VersionID`, an updated timestamp and possibly an updated cached value. We refer to this as a *Schrödinger* versioning scheme: each time we observe an `ExternalVersion` it changes. This allows Ground to track the history of an external object *as perceived* by Ground-enabled applications.

3.3.2 Model Graphs: Application Context

The model graph level of Common Ground provides a model-agnostic representation of application metadata: the *A* of our ABCs (Figure 4.) We use a graph model for flexibility: graphs can represent metadata entities and relationships from semistructured (JSON, XML) and structured (Relational, Object-Oriented, Matrix) data

models. A simple graph model enables the agility of schema-on-use at the metadata level, allowing diverse metadata to be independently captured as ad hoc model graphs and integrated as needed over time.

The model graph is based on an internal superclass called `Item`, which is simply a unique ID that can be associated with a `VersionHistoryDAG`. Note that an `Item` is intrinsically immutable, but can capture change via its associated `VersionHistoryDAG`: a fresh `Version` of the `Item` is created whenever a `Tag` is changed.

Ground's public API centers around three core object classes derived from `Item`: `Node`, `Edge`, and `Graph`. Each of these subclasses has an associated subclass in the version graph: `NodeVersion`, `EdgeVersion` and `GraphVersion`. `Node`s and `Edge`s are highlighted in the middle layer of Figure 2, with the `Node`s projected visually onto their associated versions in the other layers.

The version graph allows for ad hoc `Tag`s, but many applications desire more structure. To that end, the model graph includes a subclass of `Item` called `Structure`. A `Structure` is like a schema: a set of `Tag`s that must be present. Unlike database schemas, the `Structure` class of Ground is versioned, via a `StructureVersion` subclass in the version graph. If an `Item` is associated with a `Structure`, each `Version` of the `Item` is associated with a corresponding `StructureVersion` and must define those `Tag`s (along, optionally, with other ad hoc `Tag`s.) Together, `Tag`s, `Structure`s and `StructureVersion`s enable a breadth of metadata representations: from unstructured to semi-structured to structured.

Superversions: An Implementation Detail. The Common Ground model captures versions of relationships (e.g., `Edge`s) between versioned objects (e.g., `Node`s). The relationships themselves are first-class objects with identity and tags. Implemented naively, the version history of relationships can grow in unintended and undesirable ways. We address that problem by underlaying the logical Common Ground model with a physical compression scheme combined with lazy materialization of logical versions.

Consider updating the current version of a central `Node` M with n incident `Edge`s to neighbor `Node`s. Creating a new `NodeVersion` for M implicitly requires creating n new `EdgeVersion`s, one for each of the n incident edges, to capture the connection between the new version of M and the (unchanged!) versions of the adjacent nodes. More generally, the number of `EdgeVersion`s grows as the product of node versioning and node fanout.

We can mitigate the version factor by using a *superversion*, an implementation detail that does not change the logical metamodel of Common Ground. In essence, superversions capture a compressed set of contiguous `NodeVersion`s and their common adjacencies. If we introduce $k-1$ changes to version v of node M before we change any adjacent node, there will be k logical `EdgeVersion`s connecting M to each of its neighbor `NodeVersion`s. Rather than materializing those `EdgeVersion`s, we can use a superversion capturing the relationship between each neighbor and the range $[v, vk]$ (Figure 5). The actual logical `EdgeVersion`s can be materialized on demand by the Ground runtime. More generally, in a branching version history, a superversion captures a growing rooted subgraph of the `Version`s of one `Item`, along with all adjacent `Version`s. A superversion grows monotonically to encompass new `Version`s with identical adjacencies. Note that the superversion represents both a supernode *and the adjacent edges to common nodes*; directionality of the adjacent edges is immaterial.

3.3.3 Lineage Graphs: Behavior

The goal of the lineage graph layer is to capture usage information composed from the nodes and edges in the model graph (Figure 6.)

```
1  public class Item<T extends Version> {
2      private String id;
3  }

5  public class NodeVersion extends RichVersion {
6      // the id of the Node containing this Version
7      private String nodeId;
8  }

10 public class EdgeVersion extends RichVersion {
11     // the id of the Edge containing this Version
12     private String edgeId;
13     // the id of the NodeVersion that this EV originates from
14     private String fromId;
15     // the id of the NodeVersion that this EV points to
16     private String toId;
17 }

19 public class GraphVersion extends RichVersion {
20     // the id of the Graph that contains this Version
21     private String graphId;
22     // the list of ids of EdgeVersions in this GraphVersion
23     private List<String> edgeVersionIds;
24 }

26 public class Node extends Item<NodeVersion> {
27     // the name of this Node
28     private String name;
29 }

31 public class Edge extends Item<EdgeVersion> {
32     // the name of this Edge
33     private String name;
34 }

36 public class Graph extends Item<GraphVersion> {
37     // the name of this Graph
38     private String name;
39 }

41 public class StructureVersion extends Version {
42     // the id of the Structure containing this Version
43     private String structureId;
44     // the map of attribute names to types
45     private Map<String, GroundType> attributes;
46 }

48 public class Structure extends Item<StructureVersion> {
49     // the name of this Structure
50     private String name;
51 }
```

Figure 4: Java skeleton for Model classes.

To facilitate data lineage, Common Ground depends on two specific items— principals and workflows—that we describe here.

`Principal`s (a subclass of `Node`) represent the actors that work with data: users, groups, roles, etc. `Workflow`s (a subclass of `Graph`) represent specifications of code that can be invoked. Both of these classes have associated `Version` subclasses. Any Data Governance effort requires these classes: as examples, they are key to authorization, auditing and reproducibility.

In Ground, lineage is captured as a relationship between two `Version`s. This relationship is due to some process, either computational (a workflow) or manual (via some principal). `LineageEdge`s (purple arrows in the top layer of Figure 2) connect two or more (possibly differently-typed) `Version`s, at least one of which is a `Workflow` or `Principal` node. Note that `LineageEdge` is not a subclass of `EdgeVersion`; an `EdgeVersion` can only connect two `NodeVersion`s; a `LineageEdge` can connect `Version`s from two different subclasses, including subclasses that are not under `NodeVersion`. For example, we might want to record that Sue imported `nltk.py` in her `churn.py` script; this is captured by a `LineageEdge` between a `PrincipalVersion` (representing Sue)

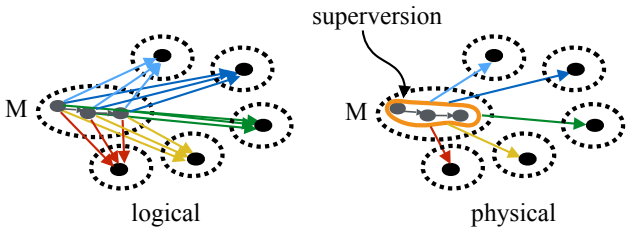

Figure 5: VersionGraph with 2 changes to a single node, in logical and physical (superversion) representations.

```
1  public class LineageEdge extends RichVersion {
2      // id of the RichVersion that this LE originates from
3      private String fromId;
4      // id of the RichVersion that this LE points to
5      private String toId;
6  }

8  public class Principal extends Node {
9  }

11 public class Workflow extends Graph {
12 }
```

Figure 6: Java skeleton for Lineage classes.

and an `EdgeVersion` (representing the dependency between the two files).

Usage data is often generated by analyzing log files, code, and/or data, and it can become very large. There are important choices about how and when to materialize lineage that are best left to aboveground applications. For example, in a pure SQL environment, the lineage of a specific tuple in a table might be materialized physically on demand as a tree of input tuples, but the lineage for all tuples in the table is more efficiently described logically by the SQL query and its input tables. The Common Ground metamodel can support both approaches depending on the granularity of `Item`s, `Version`s and `LineageEdge`s chosen to be registered. Ground does not dictate this choice; it is made based on the context information ingested and the way it is used by aboveground applications.

3.3.4 *Extension Libraries*

The three layers of the Ground metamodel are deliberately general-purpose and non-prescriptive. We expect aboveground clients to define custom `Structure`s to capture reusable application semantics. These can be packaged under `Node`s representing shared libraries—e.g., a library for representing relational database catalogs, or scientific collaborations. `StructureVersion`s allow these to be evolved over time in an identifiable manner.

3.4 `Grit`: An Illustrative Example

To demonstrate the flexibility and expected usage of our model, we discuss an aboveground service we built called Grit: the Ground-git tracker. Grit maps metadata from git repositories into Ground, allowing users to easily associate contextual information about code (e.g., wrangling or analysis scripts) with metadata about data (the inputs and outputs of the script).

Consider a git repository on Github, such as `ground-context/ground`. This repository's identity is represented by a `Node` in Ground (the central black circle in Figure 7), which we will call R for the sake of discussion. Every time a developer commits changes to the repository, git generates a unique hash that corresponds to a new version of the repository. Each one of these versions will be associated with a `Structure` that specifies two tags, a `commitHash` and a `commitMessage`

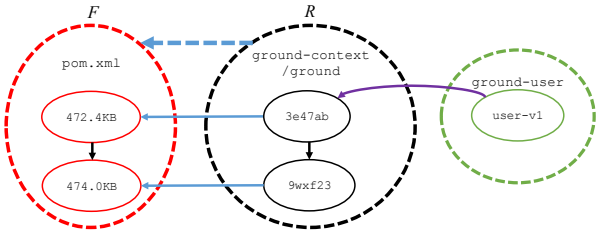

Figure 7: An illustration of Grit metadata. The `Node`s and `Edge`s have dotted lines; the `Version`s have solid lines.

(not pictured); the `Structure` ensures that every version of R will have both those tags. Grit registers a hook with Github to be notified about git commits and their hashes. Upon being notified of a new commit hash, grit calls a Ground API to register the new version of R; Ground stores this as a `NodeVersion` associated with R, containing `commitHash` and `commitMessage` tags. The API also allows grit to specify the commit hashes that preceded this new version; Ground internally relates each `NodeVersion` to its predecessor(s) via `VersionSuccessor`s (the black vertical arrows in Figure 7). Note that aboveground applications do not explicitly create `VersionSuccessor`s; the Ground API for registering a new `NodeVersion` and its parent(s) captures information that Ground uses to generate the `VersionSuccessor`s internally.

To extend the example, Grit can be augmented to track files within the repositories. Grit represents each file in Ground via a `Node`, with an `Edge` between each file and the repository `Node`. Upon hearing of a commit from Github, Grit interrogates git to determine which files changed in that commit. For a given file F that has changed (the large left oval in Figure 7), Grit creates a new `NodeVersion` (the smaller red ovals within the larger one) with metadata about the file (e.g., a size and checksum, not shown). Moreover, a new `EdgeVersion` (solid blue arrow in Figure 7) associates the new file version with the new repository version. Last, we can model users (the right, green circle in Figure 7) and the actions they perform. Once more, each user will be represented by a `Node`, which will be updated whenever the attributes of the user change—normally, not very often. There is a `LineageEdge` (the purple arrow in Figure 7) that represents the changes that a user has caused in the repository. Each `LineageEdge` points from the user `NodeVersion` to the repository commit `NodeVersion`, capturing the state of the user and the repository upoon their git commit.

The initial Grit example was chosen to be simple but useful; it can be extended to capture more aspects of git's metadata whenever such detail is deemed useful to integrate with other context in Ground. Moving beyond git, we believe that the generality of the Common Ground metamodel will allow users to capture a wide variety of use cases. In addition to git, we have currently developed basic extension libraries that allow Ground to capture relational metadata (e.g., from the Hive metastore) and file system metadata (e.g., from HDFS). We hope that more contributions will be forthcoming given the simplicity and utility of the Common Ground model.

4. GROUND 0

Our initial Version 0 of Ground implements the Common Ground metamodel and provides REST APIs for interaction with the system. Referring back to Figure 1, Ground 0 uses Apache Kafka as a queuing service for the APIs, enabling underground services like Crawling and Ingestion to support bulk loading via scalable queues, and aboveground applications to subscribe to events and register additional context information. In terms of the underground services,

Ground 0 makes use of LinkedIn's Gobblin system for crawling and ingest from files, databases, web sources and the like. We have integrated and evaluated a number of backing stores for versioned storage, including PostgreSQL, Cassandra, TitanDB and Neo4j; we report on results later in this section. We are currently integrating ElasticSearch for text indexing and are still evaluating options for ID/Authorization and Workflow/Scheduling.

To exercise our initial design and provide immediate functionality, we built support for three sources of metadata most commonly used in the Big Data ecosystem: file metadata from HDFS, schemas from Hive, and code versioning from git. To support HDFS, we extended Gobblin to extract file system metadata from its HDFS crawls and publish to Ground's Kafka connector. The resulting metadata is then ingested into Ground, and notifications are published on a Kafka channel for applications to respond to. To support Hive, we built an API shim that allows Ground to serve as a drop-in replacement for the Hive Metastore. One key benefit of using Ground as Hive's relational catalog is Ground's built-in support for versioning, which—combined with the append-only nature of HDFS—makes it possible to *time travel* and view Hive tables as they appeared in the past. To support git, we have built crawlers to extract git history graphs as `ExternalVersion`s in Ground. These three scenarios guided our design for Common Ground.

Having initial validation of our metamodel on a breadth of scenarios, our next concern has been the efficiency of storing and querying information represented in the Common Ground metamodel, given both its general-purpose model graph layer, and its support for versioning. To get an initial feeling for these issues, we began with two canonical use cases:

Proactive Impact Analysis. A common concern in managing operational data pipelines is to assess the effects of a code or schema change on downstream services. As a real-world model for this use case, we took the source code of Apache Hadoop and constructed a dependency graph of file imports that we register in Ground. We generate an impact analysis workload by running transitive closure starting from 5,000 randomly chosen files, and measuring the average time to retrieve the transitively dependent file versions.

Dwell Time Analysis. In the vein of the analysis pipeline Sue manages in Section 2, our second use case involves an assessment of code versions on customer behavior. In this case, we study how user "dwell time" on a web page correlates with the version of the software that populates the page (e.g., personalized news stories). We used a sizable real-world web log [32], but had to simulate code versions for a content-selection pipeline. To that end we wanted to use real version history from git; in the absence of content-selection code we used the code repository for the Apache httpd web server system. Our experiment breaks the web log into sessions and artificially maps each session to a version of the software. We run 5,000 random queries choosing a software version and looking up all of its associated sessions.

While these use cases are less than realistic both in scale and in actual functionality, we felt they would provide simple feasibility results for more complex use cases.

4.1 Initial Experiences

To evaluate the state of off-the-shelf open source, we chose leading examples of relational, NoSQL, and graph databases. All benchmarks were run on a single Amazon EC2 `m4.xlarge` machine with 4 CPUs and 16GB of RAM. Our initial goal here was more experiential than quantitative—we wanted to see if we could easily get these systems to perform adequately for our use cases and if not, to call more attention to the needs of a system like Ground. We acknowledge that with further tuning, these systems might perform better than they did in our experiments, though we feel these experiments are rather fundamental and should not require extensive tuning.

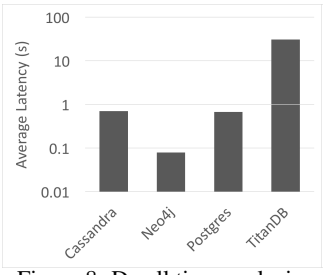

Figure 8: Dwell time analysis.

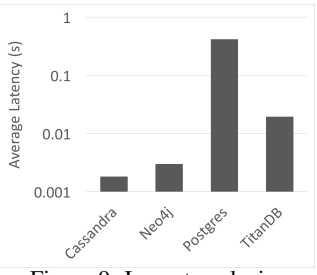

Figure 9: Impact analysis.

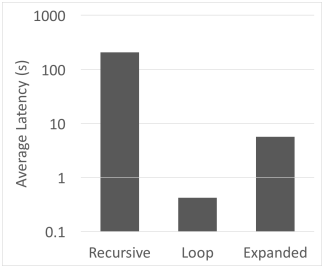

Figure 10: PostgreSQL transitive closure variants.

PostgreSQL. We normalize the Common Ground entities (`Item`, `Version`, etc.) into tables, and the relationships (e.g., `EdgeVersion`) into tables with indexes on both sides. The dwell time analysis amounts to retrieving all the sessions corresponding to a server version; it is simply a single-table look-up through an index. The result set was on the order of 100s of nodes per look-up.

For the impact analysis experiment, we compared three PostgreSQL implementations. The first was a `WITH RECURSIVE` query. The second was a UDF written in `PGPLSQL` that computed the paths in a (semi-naïve) loop of increasing length. The last was a fully-expanded 6-way self-join that computed the paths of the longest possible length. Figure 10 compares the three results; surprisingly, the UDF loop was faster than the native SQL solutions. Figure 9 shows that we were unable to get PostgreSQL to be within an order of magnitude of the graph processing systems.

Cassandra. In Cassandra, every entity and relationship from the Common Ground model is represented as a key/value pair, indexed by key. The Cassandra dwell time analysis query was identical to the Postgres query: a single table look-up which was aided by an index. Cassandra doesn't support recursive queries; for impact analysis, we wrapped Cassandra with JGraphT, an in-memory Java graph-processing library. We did not count the time taken to load the graph into JGraphT from Cassandra, hence Figure 9 shows a very optimistic view of Cassandra's performance for this query.

Neo4j. Neo4j is a (single-server) graph database, so modeling the Common Ground graphs was straightforward. The average Neo4j dwell time analysis was fast; the first few queries were markedly slow ($\sim$10 seconds), but subsequent queries were far faster, presumably due to caching. Neo4j excelled on transitive closure, performing only 50% slower than in-memory JGraphT.

TitanDB. TitanDB is a scale-out graph database designed to run over a NoSQL database like Cassandra, which is how we deployed it in our experiments on a single machine. Once again, mapping our graph-based model into TitanDB was straightforward. TitanDB's dwell time analysis performance was significantly slower than the rest of the systems, despite indexing. The impact analysis query was

significantly faster than any Postgres implementation but was still an order of magnitude slower than Neo4j and JGraphT.

Performance across systems differs significantly in our simple dwell time analysis lookups, but eve bigger divergence is seen in the impact analysis workload. We can expect impact analysis to traverse a small subgraph within a massive job history. Queries on small subgraphs should be very fast—ideally as fast as an in-memory graph system [23]. JGraphT-over-Cassandra and Neo4j provide a baseline, though neither solution scales beyond one server. PostgreSQL and TitanDB do not appear to be viable even for these modest queries. Of these systems, only Cassandra and TitanDB are designed to scale beyond a single server.

5. RELATED WORK

Work related to this paper comes in a variety of categories. A good deal of related work is based out of industry and open source, and not well documented in the research literature; for these projects we do not provide citations, but a web search should be sufficient to locate code repositories or product descriptions.

Classic commercial Master Data Management and ETL solutions were not designed for schema-on-use or agility. Still, they influenced our thinking and many of their features should be supported by Ground [21]. The Clio project is a good example of research work on this class of schema-centric data integration [25]. Much of this work could sit in aboveground application logic and be integrated with other forms of data context.

Closer to our vision are the repository systems explored in the 1990's [3]. Those systems are coupled to the programming movements of their time; for example, Microsoft Repository's primary technical goal is to "fit naturally into Microsoft's existing object architecture, called the Component Object Model (COM)" [2]. While our explicit goal here is to avoid prescribing a specific modeling framework, a number of the goals and technical challenges of those efforts presage our discussion here, and it is useful to have those systems as a point of reference.

Two emerging metadata systems are addressing governance for the open-source Big Data stack: Cloudera Navigator and the Hortonworks-led Apache Atlas. Both provide graph models that inspired Common Ground's application context model graph. They are both focused on the specifics of running jobs in today's Hadoop stacks, and provide relatively prescriptive metamodels for entities in those stacks. They do not provide versioning or provisions for integration with code repositories, and neither is perceived as vendor-neutral. LinkedIn WhereHows, FINRA Herd and Google Goods [16] are metadata services built to support the evolving data workflows in their respective organizations. Goods is a particularly well-documented and mature system, bundling what we call underground services with various proprietary services we might describe as aboveground applications.

There are a number of projects that bundle prescriptive models of metadata into interesting aboveground application use cases. OpenChorus and LabBook [19] provide portals for collaboration on data projects, including user interfaces and backing metamodels. Like many of the systems mentioned above, LabBook also uses a graph data model, with specific entities and relationships that capture its particular model of data collaboration. Vistrails [7] is a scientific workflow and provenance management system that shares some of the same goals, with a focus on scientific reproducibility. These systems are designed for particular use cases, and differ fundamentally from Ground's goals of being a standalone context management system in a decoupled stack. However these systems provide a range of examples of the kind of aboveground applications that Ground should support naturally. There has recently been a great deal of uptake in data science "notebook" tools modeled on Mathematica's Notebook—this includes the Jupyter and Zeppelin projects. Various collaborative versions of these notebooks are under development from open source and commercial groups, but the focus seems to be on collaborative editing; rich integration with a data context system like Ground could be quite interesting.

DataHub [4] is a research project that offers hosted and versioned storage of datasets, much like GitHub hosts code. Most of the DataHub research has focused on git-style checkout/checkin versioning of relational tables (e.g., [22]). Those ideas may be useful in the design of a new storage system for the versioned information that Ground needs to store, though it remains unclear if their specific versioning model will serve our general needs. A very recent technical report on ProvDB [24] echoes some of Ground's vision of coupling versioning and lineage, and proposes an architectural shim on top of a versioned store like git or DataHub. ProvDB proposes a flexible graph database for storage, but provides a somewhat prescriptive metamodel for files, actions on files, and so on. ProvDB also proposes schemes to capture activities automatically from a UNIX command shell. In this it is similar to projects like Burrito [15] and ReproZip [9].

Ground differs from the above systems in the way it factors out the ABCs of data context in a simple, flexible metamodel provided by a standalone service. Most of the other systems either limit the kind of context they support, or bundle context with specific application scenarios, or both. A key differentiator in Common Ground is the effort to be model-agnostic with respect to Application metadata: unlike many of the systems above, the Common Ground metamodel does not prescribe a specific data model, nor declare specific entity types and the way they should be represented. Of course many of the individual objectives of Ground do overlap in one way or another with the related work above, and we plan to take advantage of good ideas in the open literature as the system evolves in open source.

There is a broad space of commercial efforts that hint at the promise of data context—they could both add context and benefit from it. This category includes products for data preparation, data cataloging, collaboration, query and workflow management, information extraction, ontology management, etc. Rather than attempting to enumerate vendors and products, we refer the reader to relevant market studies, like Dresner's Wisdom of Crowds survey research [12, 11], or reports from the likes of Gartner [31, 29, 13].

The frequent use of graph data models in these systems raises the specter of connections to the Semantic Web. To be clear, the Common Ground metamodel (much like the metamodels of the other systems mentioned above) is not trying to represent a knowledge graph per se; it does not prescribe RDF-like triple formats or semantic meanings like "subjects", "predicates" or "objects". It is much closer to a simple Entity-Relationship model: a generic data modeling framework that can be used for many purposes, and represent metadata from various models including RDF, relational, JSON, XML, and so on.

6. FUTURE WORK

In the spirit of agility, we view Ground as a work in progress. Our intent is to keep Common Ground simple and stabilize it relatively quickly, with evolution and innovation happening largely outside the system core. A principal goal of ground is to facilitate continued innovation from the community: in systems belowground, and algorithms and interfaces aboveground.

6.1 Common Ground

Within Ground proper, we want to make it increasingly easy to

use, by offering developers higher levels of abstraction for the existing Common Ground API. One direction we envision is a library of common "design patterns" for typical data models. Many of the use cases we have encountered revolve around relational database metadata, so a design pattern for easily registering relational schemas is an obvious first step, and one that can build on our experience with our Hive metastore implementation. A related direction we hope to pursue is a more declarative interface for specifying models, involving simple relationships and constraints between collections or object classes. This would be a good fit for capturing metadata from typical database-backed applications, like those that use Object-Relational Mappings. From such high-level specifications, Ground could offer default (but customizable) logic for managing versioning and lineage.

6.2 Underground

As we emphasize above, we see open questions regarding the suitability of current database systems for the needs of data context. Our initial assessment suggests a need for more work: both a deeper evaluation of what exists and very likely designs for something new.

Part of the initial challenge is to understand relevant workloads for Ground deployments. Our examples to date are limited, but given the diverse participants in the early stage of the project we look forward to a quick learning curve. Three simple patterns have emerged from early discussion: tag (attribute) search, wrangling/analysis of usage logs, and traversal (especially transitive closure) of graphs for lineage, modeling and version history. The right solution for this initial workload today is unclear on a number of fronts. First, existing systems for the three component workloads are quite different from each other; there is no obvious single solution. Second, there are no clear scalable open source "winners" in either log or graph processing. Leading log processing solutions like Splunk and SumoLogic are closed-source and the area is not well-studied in research. There are many research papers and active workshops on graph databases (e.g., [5]), but we found the leading systems lacking. Third, we are sensitive to the point that some problems—especially in graphs—prove to be smaller than expected, and "database" solutions can end up over-engineered for most use cases [23].

Another challenge is our desire to maintain version history over time. Interest in no-overwrite databases has only recently reemerged, including DataHub [4] and the Datomic, Pachyderm and Noms open source systems. Our early users like the idea of versioning, but were clear that it will not be necessary in every deployment. Even when unbounded versioning is feasible it is often only worth supporting via inexpensive deep storage services. As a result, we cannot expect to provide excellent performance on general ad-hoc temporal queries; some tradeoffs will have to be made to optimize for common high-value usage.

A cross-cutting challenge in any of these contexts is the consistency or transactional semantics across underground subsystems. This is particularly challenging if databases or indices are federated across different components.

6.3 Aboveground

There is a wide range of application-level technology that we would like to see deployed in a common data context environment.

Context Extraction. One primary area of interest is extracting context from raw sources. Schema extraction is one important example, in a spectrum from automated techniques [6, 1] to human-guided metadata wrangling [18, 20]. Another is entity extraction and resolution from data, and the broader category of knowledge-base construction; examples citations here include DeepDive [27] and YAGO [33]. Turning from data to code, work on extracting data lineage is broad and ranges from traditional database provenance in SQL [8] to information flow control in more imperative languages [26] to harnesses for extracting behavior from command-line workflows [15, 9]. All of these technologies can provide useful data context in settings where today there is none; some of these techniques should be designed to improve if trained on context from other applications.

User Exhaust. The above are all explicit efforts to drive context extraction "bottom-up" from raw sources. However we suspect that the most interesting context comes from users solving specific problems: if somebody spends significant time with data or code, their effort is usually reflecting the needs of some high-value application context. Thus we're very excited about capturing "exhaust" from data-centric tools. Tools for data wrangling and integration are of particular interest because they exist in a critical stage of the data lifecycle, when users are raising the value of "raw" data into a meaningful "cooked" form that fits an application context that may be otherwise absent from the data. Notebooks for exploratory data analysis provide similar context on how data is being used, particularly in their native habitat of technical environments with relatively small datasets. Visualization and Business Intelligence tools tend to work with relatively refined data, but can still provide useful clues about the ways in which data is being analyzed—if for no other purpose than to suggest useful visualizations to other users.

Socio-Technical Networks. In all of these "data exhaust" cases, there is a simple latent usage relationship: the linkage between a users, applications and datasets. We hypothesize that tracking the network of this usage across an organization can provide significant insights into the way an organization functions—and ways it can be improved. We are not the first to suggest that the socio-technical network behavior of a data-centric organization has value (see, e.g., collaborative visual analysis [36, 37]). Yet to date the benefits of "collective intelligence" in data organizations have not been widespread in software. It is an open question why this is this case. One possibility is scale—we have yet to observe deployments where there is enough recorded data usage to produce a signal. This should be improving quickly. Another is the historically siloed nature of application context, which a service like Ground can improve. Finally, we are only now seeing the widespread deployment of intelligent applications that can actually surface the value of context: e.g. to suggest data sets, transformations or visualizations.

Ground is an environment not only to collect data context, but to offer it up via a uniform API so applications can demonstrate its utility. We believe this can be a virtuous cycle, where innovative applications that are "good citizens" in generating contextual metadata will likely benefit from context as well.

In addition, the socio-technical network around data may help redefine organizational structure and roles, as emergent behavior is surfaced. For example, it is natural to assume that data curators will surface organically around certain data sets, and official responsibilities for data curation will flow from natural propensities and expertise. Similar patterns could emerge for roles in privacy and security, data analysis and data pipeline management—much as communities form around open source software today. In a real sense, these emergent socio-technical networks of data usage could help define the organizational structures of future enterprises.

Governance and Reproducibility. Data governance sounds a bit drab, but it is critical to organizations that are regulated or otherwise responsible to data producers—often individual members of society with little technical recourse. Simple assurances like enforcing access control or auditing usage become extremely complex for organizations that deploy networks of complex software across mul-

tiple sites and sub-organizations. This is hard for well-intentioned organizations, and opaque for the broader community. Improvements to this state of practice would be welcome on all fronts. To begin, contextual information needs to be easy to capture in a common infrastructure. Ground is an effort to enable that beginning, but there is much more to be done in terms of capturing and authenticating sufficient data lineage for governance—whether in legacy or de novo systems.

Closer to home in the research community, apparently simple tasks like reproducing purely software-driven experiments prove increasingly difficult. We of course have to deal with versions of our own software as well as language libraries and operating systems. Virtualization technologies like containers and virtual machines prevent the need to reproduce hardware, but add their own complexities. It is a wide open question how best to knit together all the moving parts of a software environment for reproducibility, even using the latest tools: software version control systems like git, container systems like docker, virtual machines, and orchestration languages like Kubernetes—not to mention versioned metadata and data, for which there are no popular tools yet. We hope Ground can provide a context where these systems can be put together and these issues explored.

Managing Services That Learn. Services powered by machine learning are being widely deployed, with applications ranging from content recommendation to risk management and fraud detection. These services depend critically on up-to-date data to train accurate models. Often these data are derived from multiple sources (e.g., click streams, content catalogs, and purchase histories). We believe that by connecting commonly used modeling frameworks (e.g., scikit-learn and TensorFlow) to Ground we will be able to help developers identify the correct data to train models and automatically update models as new data arrives. Furthermore, by registering models directly with Ground, developers will be able to get help tracking and addressing many of the challenges in production machine learning outlined by Sculley et al. in [30], which focus in large part on dependencies across data, code and configuration files.

Once deployed, machine learning services are notoriously difficult to manage. An interesting area of future research will be connecting prediction serving platforms (e.g., Velox [10] and TensorFlow Serving [34]) to Ground. Integration with Ground will enable prediction serving systems to attribute prediction errors to the corresponding training data and improve decision auditing by capturing the context in which decisions were made. Furthermore, as prediction services are composed (e.g., in predicting musical genres and ranking songs), Ground can provide a broader view of these services and help to isolate failing models.

7. CONCLUSION

Data context services are a critical missing layer in today's Big Data stack, and deserve careful consideration given the central role they can play. They also raise interesting challenges and opportunities spanning the breadth of database research. The basic design requirements—model-agnostic, immutable, scalable services—seem to present new database systems challenges underground. Meanwhile the aboveground opportunities for innovation cover a broad spectrum from human-in-the-loop applications, to dataset and workflow lifecycle management, to critical infrastructure for IT management. Ground is a community effort to build out this roadmap—providing useful open source along the way, and an environment where advanced ideas can be explored and plugged in.

Acknowledgments

Thanks to Alex Rasmussen for feedback on early drafts of this paper, and to Hemal Gandhi for input on Common Ground APIs and supernodes. Thanks also to Frank Nothaft for ideas and perspective from biosciences, and to David Patterson for early support of the project. This work was supported in part by a grant from the National Institutes of Health 5417070-5500000722.

8. REFERENCES

[1] M. D. Adelfio and H. Samet. Schema extraction for tabular data on the web. *Proceedings of the VLDB Endowment*, 6(6):421–432, 2013.

[2] P. A. Bernstein, T. Bergstraesser, J. Carlson, S. Pal, P. Sanders, and D. Shutt. Microsoft repository version 2 and the open information model. *Information Systems*, 24(2):71–98, 1999.

[3] P. A. Bernstein and U. Dayal. An overview of repository technology. In *VLDB*, volume 94, pages 705–713, 1994.

[4] A. 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.

[5] P. Boncz and J. Larriba-Pey, editors. *International Workshop on Graph Data Management Experiences and Systems (GRADES)*. ACM, 2016.

[6] M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. WebTables: Exploring the power of tables on the web. *Proceedings of the VLDB Endowment*, 1(1):538–549, 2008.

[7] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. VisTrails: Visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 745–747. ACM, 2006.

[8] J. Cheney, L. Chiticariu, and W.-C. Tan. *Provenance in Databases*. Now Publishers Inc, 2009.

[9] F. S. Chirigati, D. E. Shasha, and J. Freire. ReproZip: Using provenance to support computational reproducibility. In *Workshop on the Theory and Practice of Provenance (TaPP)*, 2013.

[10] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems*, 2015.

[11] H. Dresner. 2016 Collective Insights Market Study Report. Technical report, Dresner Advisory Services, LLC, 2016.

[12] H. Dresner. 2016 End User Data Preparation Market Study. Technical report, Dresner Advisory Services, LLC, 2016.

[13] A. D. Duncan, D. Laney, and G. D. Simoni. How chief data officers can use an information catalog to maximize business value from information assets. Technical report, Gartner, Inc., 2016.

[14] Gartner. Gartner says every budget is becoming an IT budget, Oct. 2012. http://www.gartner.com/newsroom/id/2208015.

[15] P. J. Guo and M. Seltzer. BURRITO: Wrapping your lab notebook in computational infrastructure. In *Workshop on the Theory and Practice of Provenance (TaPP)*, 2012.

[16] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing Google's datasets. In *Proceedings of the 2016 International Conference on Management of Data*, pages 795–806. ACM, 2016.

[17] W. H. Inmon. *Building the data warehouse*. John Wiley & Sons, 2005.

[18] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3363–3372. ACM, 2011.

[19] E. Kandogan, M. Roth, P. Schwarz, J. Hui, I. Terrizzano, C. Christodoulakis, and R. J. Miller. LabBook: Metadata-driven social collaborative data analysis. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 431–440. IEEE, 2015.

[20] V. Le and S. Gulwani. FlashExtract: A framework for data extraction by examples. In *ACM SIGPLAN Notices*, volume 49, pages 542–553. ACM, 2014.

[21] D. Loshin. *Master Data Management*. Morgan Kaufmann, 2010.

[22] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. Parameswaran, and A. Deshpande. Decibel: The relational dataset branching system. *Proceedings of the VLDB Endowment*, 9(9):624–635, 2016.

[23] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *HotOS XV*, 2015.

[24] H. Miao, A. Chavan, and A. Deshpande. ProvDB: A system for lifecycle management of collaborative analysis workflows. *arXiv preprint arXiv:1610.04963*, 2016.

[25] R. J. Miller, M. A. Hernández, L. M. Haas, L.-L. Yan, C. H. Ho, R. Fagin, and L. Popa. The Clio project: Managing heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

[26] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241. ACM, 1999.

[27] F. Niu, C. Zhang, C. Ré, and J. W. Shavlik. DeepDive: Web-scale knowledge-base construction using statistical learning and inference. *VLDS*, 12:25–28, 2012.

[28] D. Patil. *Data Jujitsu: The Art of Turning Data into Product*. O'Reilly Media, 2012.

[29] R. L. Sallam, P. Forry, E. Zaidi, and S. Vashisth. Market Guide for Self-Service Data Preparation. Technical report, Gartner, Inc., 2016.

[30] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[31] G. D. Simoni and R. Edjlali. Magic Quadrant for Metadata Management Solutions. Technical report, Gartner, Inc., 2016.

[32] Star Wars Kid: The Data Dump, 2008. http://waxy.org/2008/05/star_wars_kid_the_data_dump/, retrieved June, 2008.

[33] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706. ACM, 2007.

[34] TensorFlow Serving. https://tensorflow.github.io/serving.

[35] The Basel Committee. The Basel iii Accord, 2016. http://www.basel-iii-accord.com, retrieved November, 2016.

[36] F. B. Viegas, M. Wattenberg, F. Van Ham, J. Kriss, and M. McKeon. ManyEyes: A site for visualization at internet scale. *IEEE transactions on visualization and computer graphics*, 13(6):1121–1128, 2007.

[37] W. Willett, J. Heer, J. Hellerstein, and M. Agrawala. CommentSpace: Structured support for collaborative visual analysis. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 3131–3140. ACM, 2011.