Establishing Common Ground with Data Context

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

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

ABSTRACT

Ground is an open-source *data context service*; a system to manage all the peripheral 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 along three axes: *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. Throughout, 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."* [4] 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 [3]. The changes in economics and usage brought on the "three V's" 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 [7]. 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.

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

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

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

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 V’s” 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—transformation 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 logs and sensors). Not only must we track upstream lineage— the data sets and code that led to the creation of the data object—but we must also track the downstream lineage, including the data products that were 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 and associated code, including changes over time to both structure and content. Traditional metadata focused on the present, but historical context is increasingly useful in modern 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. 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. GROUND: SCENARIOS AND DESIGN

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.

To illustrate the potential of the Ground data context service, we describe a concrete scenario in which Ground is used to 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 the scenario, we explore the design requirements for a data context service.

2.1 Scenario: Context-Enabled Analytics

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 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 transformation scripts and suggests unnesting attributes and pivoting them into tables. Based on code and usage information in Ground, Preply then suggests applying a sentiment analysis package that is widely used at the bank. 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 transformation scripts registered with Ground by other analysts to suggest standardized join keys 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 Design Requirements

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. To this end 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 3.2.

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 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 [2]. 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.

3. ARCHITECTURE OF GROUND

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.

3.1 Key Services

Ground's functionality is backed by five decoupled subservices. For agility, we are starting the project using existing open source solutions for each. 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 or in batches. The main issue 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. 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 DeepDive, 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 open-source DBMSs. As noted there, we believe this is an area for significant database research.

Search and Analyze. 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. Second, intelligent applications like those in Section 2.1 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.

Identity 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,

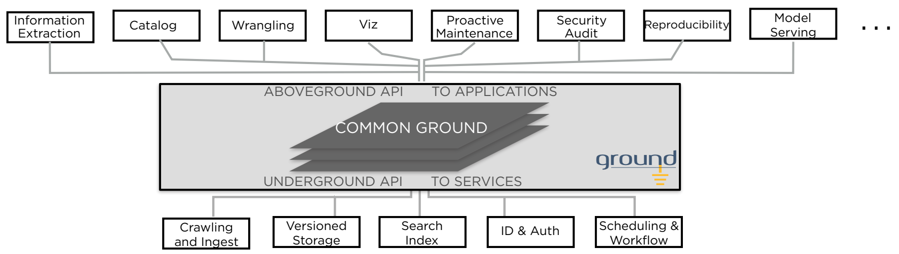

Figure 1: The initial architecture of Ground.

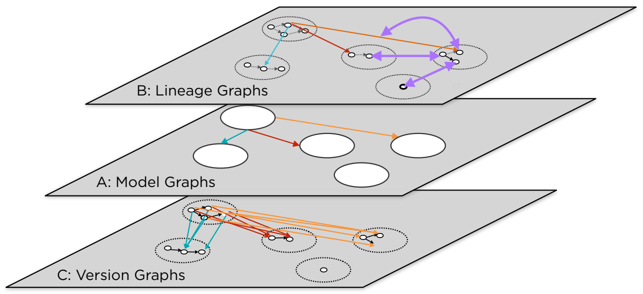

Figure 2: The Common Ground metamodel.

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 [8] 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.

3.2 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.

3.2.1 Version Graphs: Representing Change

We begin with the version graph layer of Common Ground, which the *C* in the ABCs of data context. 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 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 depicted by the small circles in the bottom layer of Figure 2. `Version`s can be linked 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.

To allow customization, the version graph allows `Version` objects to be associated with ad hoc `Tag`s (key-value pairs) upon creation. Both `Version`s and `Tag`s are themselves immutable.

3.2.2 Model Graphs: Application Context

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

External Items and Schrödinger Versioning
We often wish to track items whose metadata is managed outside of Ground: canonical examples include GitHub repositories and Google Docs. Ground cannot automatically track these items as they change. They are represented in Ground by `ExternalItem`s, with `ExternalVersion`s that contain various tags: a `timestamp`, optional `cachedValue`, and parameters for accessing the reference (e.g., port, protocol, URI, etc.) Whenever a Ground client uses the Aboveground API to access an a `ExternalVersion`, this results in Ground fetching the object and generating 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 `ExternalItem` it changes. This allows Ground to track the history of an external object *as perceived* by Ground-enabled applications.

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

3.2.4 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.

4. GROUND v0

Our initial version of Ground implements the Common Ground metamodel and provides REST APIs for interaction with the system. It currently makes use of LinkedIn Gobblin for crawling and ingest and Apache Kafka for queueing and pub/sub. 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 Authorization and 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 them 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 perform impact analysis 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 2.1, 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 (http://waxy.org/2008/05/star_wars_kid_the_data_dump/) 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 Apache httpd web server repository. 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 intensive tuning.

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 5 compares the three results; surprisingly, the UDF loop was faster than the native SQL solutions. Figure 4 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

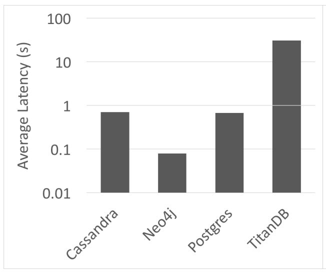

Figure 3: Dwell time analysis.

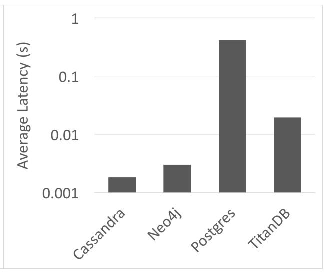

Figure 4: Impact analysis.

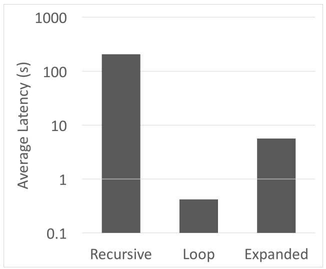

Figure 5: PostgreSQL transitive closure variants.

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 4 shows a very optimistic view of Cassandra's performance for this query.

Neo4j. Neo4j is a (single-node) 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.

While there is variance in our simple dwell time analysis lookups, the bigger divergence is 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 [6]. JGraphT-over-Cassandra and Neo4j provide a baseline, though neither solution scales beyond one node. 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 node.

5. RELATED WORK

Work related to this paper comes in a variety of categories. For submission brevity, we focus on related system architectures and omit bibliography entries to systems easily found online. An accepted version will contain a more scholarly bibliography.

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 supportable as a subset of the Ground functionality [5]. Meanwhile, two emerging metadata systems address governance for Big Data: Cloudera Navigator and the Hortonworks-led Apache Atlas. Both provide graph models that inspired the Common Ground model graph; neither provides versioning or is perceived as vendor-neutral.

There is a broad space of efforts that illustrate the possibilities of data context. There are many commercial applications and research projects for Aboveground tasks: cataloging, wrangling, lineage analysis, query and workflow management, information extraction, ontology management, etc. OpenChorus and IBM LabBook provide overarching portals for collaboration. LinkedIn WhereHows, FINRA Herd and Google Goods are metadata services built to support data workflows in their respective organizations; Goods bundles what we call Underground services with various proprietary services we might describe as Aboveground applications. All of the above make use of specific metamodels customized to their needs. By contrast, Ground is focused on designing a reusable service to enable diverse applications and organizations to integrate their notions of data context. This shapes the core challenges and opportunities we are pursuing, and hopefully will encourage adoption and innovation.

Our initial assessment of storage engines suggests a need for more work: both a deeper evaluation of what exists and very likely designs for something new. There are many research papers and active workshops on graph databases (e.g., [1]), but we found the leading systems (Neo4j, Titan) lacking. Interest in no-overwrite databases has only recently reemerged, including the DataHub research project, and the Datomic, Pachyderm and Noms systems. The combination of graphs and immutable versions appears to be a new challenge.

6. 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.

7. REFERENCES

[1] International workshop on graph data management experiences and systems (GRADES), 2016.

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

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

[4] W. H. Inmon. *Building the data warehouse*. John wiley & sons, 2005.

[5] D. Loshin. *Master data management*. Morgan Kaufmann, 2010.

[6] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *HotOS XV*, 2015.

[7] D. Patil. *Data Jujitsu: The art of turning data into product*. O'Reilly Media, 2012.

[8] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing distributed systems with information flow control. In *NSDI*, volume 8, pages 293–308, 2008.