Depot: Dependency-Eager Platform of Transformations

Kerem Celik
UCSB Computer Science Department
Santa Barbara, CA, USA
kerem@ucsb.edu

Samridhi Maheshwari
UCSB Computer Science Department
Santa Barbara, CA, USA
samridhimaheshwari@ucsb.edu

Shereen ElSayed
UCSB Computer Science Department
Santa Barbara, CA, USA
s_elsayed@ucsb.edu

Markus Mock
HAW Landshut
Landshut, Germany
mock@haw-landshut.de

Chandra Krintz
UCSB Computer Science Department
Santa Barbara, CA, USA
ckrintz@cs.ucsb.edu

Rich Wolski
UCSB Computer Science Department
Santa Barbara, CA, USA
rich@cs.ucsb.edu

***Abstract*—This paper presents a new model for a data management system specifically designed to enable community-curated data repositories and collaboration. Depot (a Dependence-Eager Platform of Transformations) is based on a data-lake approach that eases the technological burdens associated with data contribution while providing an interactive programming environment for developing transformations that result in structured tables supporting SQL database operations. Crucially, Depot implements lazy evaluation of these transformations so that only the structured data that is demanded by a data consumer is generated. Until the structured data is "materialized," Depot tracks and maintains the dependencies that are required to perform the eventual materialization. This lazy approach to creating structured data allows Depot to maintain a smaller resource footprint compared to a typical data warehouse approach while maintaining the flexibility of the data lake model. Furthermore, Depot is designed as a community-sustainable platform. The initial prototype is implemented for cloud deployment and it distributes the storage and ETL workload cost among the data consumers. Performance results of the early prototype are encouraging, making Depot a new infrastructure for creating data lakes that foster contributed-consumer collaboration.**

***Index Terms*—data lake, cloud services for science**

I. Introduction

Community curation of data is an emerging model of collaboration that stimulates innovation through online data sharing by community members who are not necessarily working together on a common project or problem. For example, Zooniverse [24], [28] brokers cooperation between researchers and contributors: Researchers describe and share specific datasets while contributors provide insights and analyses on those datasets on a volunteer basis. Other citizen science efforts [4], [11], [13], [21] use a similar model to establish loose collaborations between contributors who may be working on a variety of separate efforts using the same "pool" of data. For software development, arguably one of the most critical community-facing sharing venues is github [17] where users share, contribute, modify, and extend software source code using the git [25] source-code control system.

These systems typically require that the community data contributor ingress the data into a repository by first transforming it so that it matches an internal (and possibly hidden) schema. Nature's Notebook [21], for example, exports a form-based interface that citizen observers use to input their observations. Alternatively, systems that do not require a transformation (e.g., GitHub) often support only limited database functionality.

For researchers, manipulating community-curated data can be an especially laborious task. Either data contribution is subject to a rigorous integrity-checking process on ingress (creating a significant transformation burden for the contributor), as in [1], or the data can be ingressed as unstructured or semi-structured data (as in [10]) making database operations challenging or infeasible.

In this paper, we propose Depot (a Dependency-Eager Platform of Transformations [1]) as a new model for a data management system specifically designed to enable community-curated data repositories and collaboration. Depot's approach is to allow the ingress of "raw" data in any form along with metadata that describes its formatting. It couples this raw data pooling with a Python-based interactive programming environment (i.e., a Jupyter notebook [7]) so that consumers of each dataset can write scripts (in Python) to create structured data tables, which Depot also manages [2]. The data tables that result from transformations support SQL database operations that can also be applied via the interactive interface. Each table that Depot manages corresponds to a tree of "upstream" raw datasets and tables, and a corresponding transformation script necessary to create the "downstream" table from the

This research was generously sponsored by VMware and NSF grant CNS-2107101.

[1]Depot is available as open source from https://github.com/MAYHEM-Lab/Depot

[2]These "ingress" scripts are typically termed "ETL" scripts which is an acronym "Extract-Transform-Load." Depot automates the extraction and loading of data, so the Depot user is only responsible for writing the transformation.

upstream dependencies. All scripts, raw datasets, and tables are immutable and versioned so that users of Depot can determine the provenance of the data in the dependency tree associated with each table, and data cannot simply "disappear" when its owner chooses to update it.

Importantly, Depot does not convert raw data to structured data when it is ingressed or when a dependency is created. Instead, datasets and dependencies are *announced* so that users can discover their existence, but structured data tables are only created when a dataset is *materialized* in response to a user command or a fixed schedule. Materializing a dataset as a table causes Depot to trace back through the dataset's dependency tree and materialize any announced but unmaterialized dependencies. In this way, Depot only "expends" the resources (database table space and Extract-Transform-Load – ETL – processing cycles) that are necessary to create structured data that users wish to manipulate.

This lazy approach to creating structured data allows Depot to maintain a smaller resource footprint than a typical data warehouse approach. If ETL were performed at data ingress (as with most data warehouses), the space necessary to hold the resulting tables and the ETL computation (which could be lengthy) would need to be committed *even if there are no consumers for the resulting table*. This expenditure is particularly costly if the data is then updated with a new version (causing another ETL cycle) before the first version is consumed (especially if the data is to be versioned).

Finally, Depot is designed as a community-sustainable platform. Its implementation is cloud-based, and it attempts to distribute the storage and ETL workload to the consumers of the data. The essential principle is that contributors should "pay" little for their contribution. The platform operator should not incur a cost because the operator does not own the data. A consumer with use for data should incur the burden of its use as expressed as a materialization.

Taken together, this set of design goals makes Depot an early (if not inaugural) attempt to create a community-sustainable platform for creating "data lakes" that fosters contributor-consumer collaboration. In this paper, we describe the abstractions that Depot defines for this purpose, discuss its implementation for cloud-based deployment, and detail its operation using a set of examples. We also present initial performance results for a prototype implementation and end with conclusions and an outlook on future work.

II. Background and Related Work

The concept of a "data lake" is originally due to James Dixon of Pentaho in opposition to a "data mart" where the data is maintained as a set of products. More recently, the contrast is with a data warehouse [14], a support infrastructure for data mining. In particular, a data warehouse notably frees its data-consuming users (but not its data contributors) from the burden of ETL activities by rationalizing the data on ingress against a predefined schema.

Data lakes typically evince several properties, such as governance and scalability while specifically supporting data exploration and discovery, and facilitating data analysis [15]. Data lake governance refers to the policies, processes, and controls implemented to ensure effective data management, security, and usage compliance within a data lake. To support governance, Hai et al. [12] study metadata management, semantics, and enrichment. The metadata is extracted from the data and used to create models and links between data, thereby improving data quality. Depot shares the notions of curated metadata and dependency links, but extends data quality management to include versioning and provenance.

Benjelloloun et al. [2] use elastic stacks for security [5]. They use Apache Ranger to implement access control policies. Depot is similar in that it enforces access control policies set by each dataset's owner and maintains these policies through dependency trees. Depot's present policy implementation mirrors that of GitHub, but richer policies are possible.

Sarramia et al. [23] postulate a similar reductionist set of access control policies in which there are three levels of visibility: open-data, which is available for everyone to access; private which limits the access to metadata and data to people working on the project but appears in the search; and embargo, which gives access to only the metadata to the people on the project and omits it from search results. The authors apply FAIR principles (Findable, Accessible, Interoperable, Reusable) [6] to their storage and redistribution mechanisms for data and metadata. Depot also implements FAIR principles, but extends them with additional principles as discussed in the next section.

III. Depot Principles and Abstractions

It is our thesis that community-sustainable data repositories are best structured as data lakes [18]. The eventual use of data may not be envisioned by its contributor (e.g., a citizen scientist) and the platform operator should play the role of a "broker" rather than an organization (e.g., a company operating a data warehouse) that owns the data that has been ingressed.

As such, Depot allows data providers to ingest their data into the system without considering hosting locations, formats, or delivery. Similarly, data consumers can use Depot to discover existing data, build dynamic data pipelines, and serve as producers by contributing their pipelines back to the system for consumption by others. Depot is a trusted arbitration platform facilitating disconnected yet collaborative data access between mutually-distrusting entities.

In addition to FAIR functionality [6], Depot adheres to the following principles.

- **Data origination and data consumption are decoupled.** The originator or contributor need not manifest (in the form of a schema) the eventual usages for the data.
- **The platform must support collaboration.** Users must be able to share the results of their efforts as a community.
- **The system should have an interactive interface.** Because the use of contributed data may not be evident when it is contributed, the system must support exploration and experimentation. This requirement militates for the ability

to use the system interactively (as opposed to strictly in a "batch" mode).

- **The platform's operations must be auditable.** Data within the system must have a discoverable chain of custody so that users can determine the provenance of the data products they produce.
- **The platform should be operationally unified.** From both the perspective of users and the platform's operator, it should function as a unified system and not an amalgamation of disparate technologies.

A. The Depot Dataset Abstraction

Depot's core abstraction is the *dataset*, which represents and encapsulates a dynamic digital asset. The dataset serves as a typed interface for discovering, consuming, transforming, and sharing assets while abstracting away the underlying storage locations, data formats, and integrations. Changes in a dataset's underlying digital asset are captured as snapshots and represented as an ordered sequence of immutable *versions*.

Depot provides a pluggable interface for defining types, or schemas, on datasets and enforces this type on all versions. Depot's logical translation layer uses these types to facilitate data access between producers and consumers operating with different data formats. For any type, conversions can be defined between underlying data formats, and, provided that the target format is reachable via a path of conversions, consumers can seamlessly interact with datasets in any format they choose. Data format translation adds computational and storage overhead. However, it obviates the need for producers and consumers to negotiate physical data formats, which in many cases also forces collaborators to be locked into mutually-compatible technologies. Without Depot as an intermediary, data consumers must adopt the data processing ecosystems their producers use or build their own complex, ad-hoc conversion mechanisms.

B. Transformation and Materialization

In addition to being ingested into Depot from an external system, datasets can be declared by defining *transformations* on a set of input datasets; the platform automatically implements the "Extraction" and "Loading" associated with ETL. New datasets can be arbitrarily derived from other datasets, allowing for elaborate, lazily-evaluated data pipelines which preserve precise data lineage and greatly reduce the barrier of entry for data producers and consumers. Transformations are an immutable set of instructions that encapsulate the logic of creating a new dataset version from versions of the input datasets. As new versions of the input datasets are made available, the transformation *announces* new versions of its target dataset but defers *materialization* until explicitly requested. An *announced* dataset version indicates that the version's underlying physical data is not yet present, but because the input versions and transformations are immutable, they can be produced at any time. This process of evaluating an announced version's input versions and the associated transformation's instructions to produce the underlying data is known as materialization.

Announcement is a recursive process enabling the creation of trees of datasets of arbitrary complexity where changes in the root datasets are recursively reflected in all downstream datasets. Materialization is similarly recursive, but in the other direction: when materialization is requested for a dataset version, the dataset tree is traversed toward the roots to materialize all announced versions that are a dependency, after which the version's transformation is evaluated, and the output is persisted.

Transformations are evaluated by Depot-controlled computational resources, providing data consumers, which may not trust providers, with a guarantee as to exactly how data was operated on and modified. The resource cost incurred by the computation to materialize a dataset version on the path is attributed to the entity that requested the materialization.

Materialization can also be explicitly requested by a consumer on a specific version, or by a *materialization schedule* set on the dataset. A materialization schedule allows a dataset owner to provide an upper bound for the tolerated staleness of the dataset by ensuring that there is always one materialized version that is not older than the latest announced segment by some configurable duration.

The ability for users to produce new datasets with transformations drastically reduces the obstacles that potential contributors may face. One who wishes to derive new datasets from existing datasets can do so by simply defining a transformation, without considering storage locations, computational systems, data formats, or synchronizing evaluation schedules. Because transformations are lazily evaluated, the owner of a dataset without an explicit materialization schedule incurs no resource cost, allowing providers to derive exploratory datasets without an explicit use in mind. Only when a consumer starts to use the dataset does Depot start to evaluate and persist data, and the resource cost of the materialization and storage is incurred by the consumers.

Because materialization is always possible once dependencies are announced, lazy evaluation of dataset versions also prevents large, complex data pipelines from suffering from unnecessary latency bottlenecks. In particular, the maximum possible frequency for dataset materialization is bounded by its parent datasets' announcement frequency. In a pipeline, intermediate datasets (or dataset versions) that are only announced are only materialized if, and when, they are needed. As long as versions of the dependency datasets are announced frequently enough, a dataset can be derived from it with minimum upstream materialization latency. By contrast in a data pipeline system without lazy evaluation, *all* versions of all datasets are materialized even when some versions are never used and a downstream materialization might need to wait for unneeded upstream processing.

C. Transformations

Transformations are parts of ETL programs that take one or more datasets (themselves either raw or transformed and structured) as inputs and produce a structured dataset as an output. Depot stores transformations as immutable objects. In this way, a user of a transformation is guaranteed to be able to produce its outputs in the future as long as the input dependency tree has not been truncated by a retention policy (see below). Users can then delete datasets that can easily be reproduced, thereby reducing the storage burden.

When a dataset is published, if it is not a raw dataset, the transformation necessary to create the dataset is also published. The access controls on the transformation are the same as those on the data itself.

D. Data Lineage

Transformations create dataset versions and each dataset version retains precise information about which versions of which datasets are contained it its dependency tree, providing an exact lineage for any piece of data. The entire lineage of a dataset version is persisted as long as the version itself is retained.

Persisting the lineage of a dataset and its versions provides transparency, giving data consumers confidence about the reliability of their data. Inspection of the transformations and input datasets that were used to produce a dataset version allow consumers to assess the quality and compliance of data quickly.

E. Retention Policies

Retention of a dataset's physical artifacts are managed via a reference-counting mechanism. A dataset version holds a reference to all parent dataset versions in its lineage. Datasets can be configured to have their versions hold either *strong* or *weak* references to parent versions. A *retention duration* may also be set on a dataset; this causes an implicit temporary strong reference to be held against new versions for a set amount of time.

If a dataset version has no strong references held against it, it is considered eligible for collection, and Depot may delete its physical assets, held references, and lineage metadata. When a version is deleted, all versions with a weak reference to it are similarly deleted.

An entity's storage utilization depends on how many strong references their dataset versions hold. This behavior enables providers and consumers with different tolerances for storage cost to effectively collaborate. A data consumer with a need to retain a lengthy history can configure their datasets as holding strong references against parents, forcing retention of parent versions without burdening the upstream providers with the cost. If the parent owner deletes the dependency, the storage burden shifts to the nearest downstream consumer. The accounting for downstream "inheritance" of strong upstream dependencies is the subject of our ongoing research. For example, if the original owner deletes the dataset, one option could be to share (evenly) the storage cost among all downstream consumers with a strong dependence. As such, a cost-conscious consumer may derive datasets that hold only weak references, creating "best-effort" datasets, versions of which exist only as long as another entity is willing to retain parent versions.

Note that retention policies must be visible to users considering creating a dependency on the associated datasets. In particular, they do not govern access control but are necessary to allow users to manage their resource expenditures. For example, a user who wishes to create a dataset dependent on an upstream dataset with a weak retention policy must decide whether the dataset created is permanent or ephemeral. If the intention is to create a dataset that is permanent, the user must materialize the dataset (before Depot reclaims its weak upstream dependencies) and set its retention policy to *strong*. As described, the newly created dataset, having a strong retention policy, is "charged" to the creating user's resource account. If the user had chosen the ephemeral option, Depot need only charge the user for the storage at the time of materialization and then only until the reclamation of the upstream dependency.

F. Access Control Policies

In principle, Depot allows the "owner" of a dataset to set its access control policies. A contributor of raw data is the raw dataset's owner. Derived datasets are owned by their deriving users, who can only set access control policies that attenuate the most restrictive access control policies associated with the dependency tree for each derived dataset.

This general access control functionality, while powerful, leads to complexities that we have yet to be able to address successfully. For example, a creator of a derived dataset must analyze the access control policies for all dependencies in the dependency tree to determine the intersection of what is permitted by the union of all policies. She must then determine an access control policy for the dataset she creates that is no more permissive.

Even when a dedicated and knowledgeable user can accomplish this task, the burden of policy attenuation on the dataset creator necessarily implies that the policies themselves must be public. If, for example, two separate policies forbid the combination of data by competing research groups, that information cannot be hidden from a user (who is part of neither group and has access to both datasets) considering using them as a dependency. Thus, in some sense, the policies must necessarily "leak" information. That is, the neutral user must be able to see that the two groups do not wish to cooperate. Further, unlike retention policies, defining a unified rights transfer policy to be invoked when an owner revokes upstream access is, at present, an unaddressed challenge.

For these reasons, the initial Depot prototype implements a simple public-private dichotomy (including group-private for collaborators with mutual trust) in which public access cannot be revoked. We plan to study enriching the policy mechanisms in Depot as part of our future work.

G. Accounting

One goal of Depot is to prevent the platform operator from taking a stake in its operation. This model contrasts with other community repositories (such as GitHub), which technically take ownership (or joint ownership) of the data they host as recompense for the hosting expenses. Except for the Depot control plane, the goal for Depot is to assign the hosting "costs" to its users according to their usage and to incentivize usage by minimizing this cost.

Depot assumes a cloud hosting model in which individual users are charged for tenancy, possibly according to organization membership. The current prototype uses Eucalyptus [8], [22] to emulate Amazon AWS on local resources but also leverages the quota features that Eucalyptus supports. In particular, each user is associated with a quota (in terms of storage footprint and ETL load). That is, a user cannot materialize a dataset with a strong retention policy unless there is sufficient quota in that user's account.

Quota transfer in response to retention policy or user-triggered deletion of an upstream dependency is more complex. If a user creates a dataset with a strong retention policy, the user must also have quota sufficient to materialize any upstream dependencies as well. In this way, if the upstream owner deletes a dataset or a retention policy times out a dataset with a weak policy, the quota for maintaining the dependency transfers to the nearest consumer with the oldest dependency.

We still need to observe the efficacy of this quota mechanism sufficiently to determine its actual utility. On the one hand, it allows Depot to make strong guarantees about materialization while transparently and predictably exposing the potential costs. On the other hand, it discourages archival activities since the effects of potential quota transfer are cumulative. We are considering a modification of this policy that would "charge" the quotas of all downstream consumers for an equal fraction of the hosting cost. While fair, this modification would mean that quota charges could fluctuate when downstream consumers delete dataset to reclaim the fraction of the quota. This alternative is attractive, however, because it incentivizes users to commit quota to a dependency in proportion to the importance of the materialization of a dataset upon which it depends.

Finally, our limited experience with Depot indicates that users materialize datasets they wish to preserve and charge these datasets to their respective quotas. When exploring, they set weak retention policies and avoid the quota charges associated with upstream dependencies.

H. Streaming Data

Depot's dataset abstraction represents structured data (e.g. data that can be queried using relational query primitives). To integrate streaming data, Depot adapts the λ-architecture [19] which combines a "speed layer" (usually a streaming database) with a "batch layer" (supporting full query semantics). Unlike a typical λ-architecture (which exports both layers to the user interface), Depot uses the speed-layer to generate announcements and/or materializations in the batch layer.

In Depot, a stream is represented as a unique Depot data type and a topic-based publish-subscribe (pub-sub) communication abstraction [16]. Users associate a Depot dataset using a Depot stream represented by a pub-sub topic, and a transformation script for materializing the dataset. When new data available becomes for that topic, a new flattened version of the stream is announced or instantly materialized depending on the user's logic.

I. Example

We next demonstrate an example use of Depot for a collaboration from the RiPiT [3] project in which three organizations are exploring energy generation and emissions data within a region. Organization A has tabular data about the greenhouse-gas (GHG) emissions of each subregion within the overall region and ingests this data into Depot as a table dataset. Organization B has a listing of all generating stations within the region, without locational data, and publishes it to Depot as an unstructured dataset. Organization C provides geospatial information about energy infrastructure within the region, such as transmission lines, distribution systems, load nodes, and generation nodes. Each organization instructs Depot that they would like to consume storage quota to retain only the three most recent versions of their respective datasets.

In this example, Organization C realizes the value of augmenting Organization B's plant listing with their geospatial data, and defines a transformation which consumes both datasets and produces a table which enumerates all generating stations and their associated geographical coordinates. Since Organization C has no immediate use for this data, they choose not to configure a materialization schedule for the dataset and consume no storage or compute resources.

Organization A is then able to apply a transformation on Organization C's new dataset and their own locational greenhouse-gas emissions table to join both, producing a table that exposes emissions over time per-generating station. The dependency graph of these organizations' datasets is depicted in Figure 1.

If Organization A and B were to each upload one version of their dataset, and Organization C were to upload 3, the dependency graph of individual dataset versions would appear as depicted in Figure 2.

Consider then, a user D, distinct from Organizations A, B, or C, which requests explicit materialization of the emissions timeseries dataset. This user requests a weak materialization of version 1 and a strong materialization of version 2. These materialization requests force materialization of versions 1 and 2, respectively, of the locational plant dataset, and results in the state depicted in Figure 3.

The weak materialization of version 1 of the emissions dataset causes user D to hold weak references to version 1 of each of the aforementioned datasets. Similarly, version 2's strong materialization results in strong references being held against version 2 of all datasets. Thus, even if the organizations produce a sufficient number of new dataset versions to cause their respective versions 1 and 2 to fall out of retention, user D is guaranteed that Depot will retain all versions that are upstream of version 2 of the emissions timeseries dataset

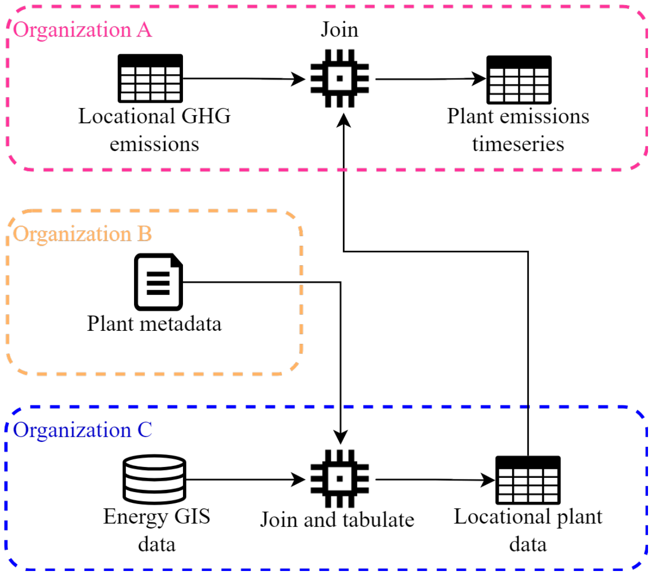

Fig. 1: Dataset and transformation dependency graph as described in the example.

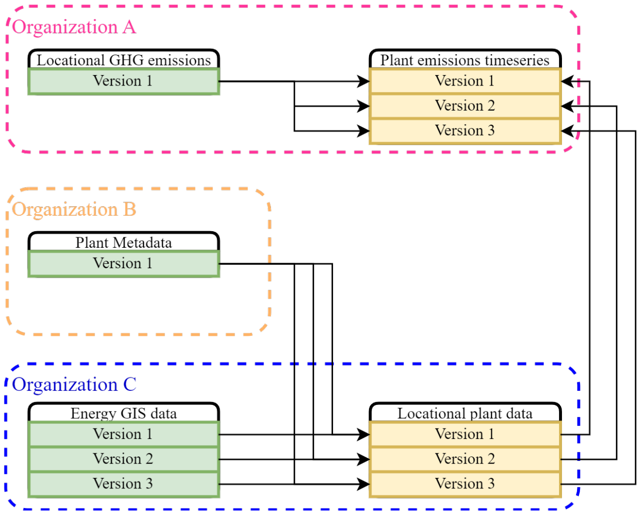

Fig. 2: Dataset version dependency graph. Announced versions are shaded yellow and materialized versions are shaded green.

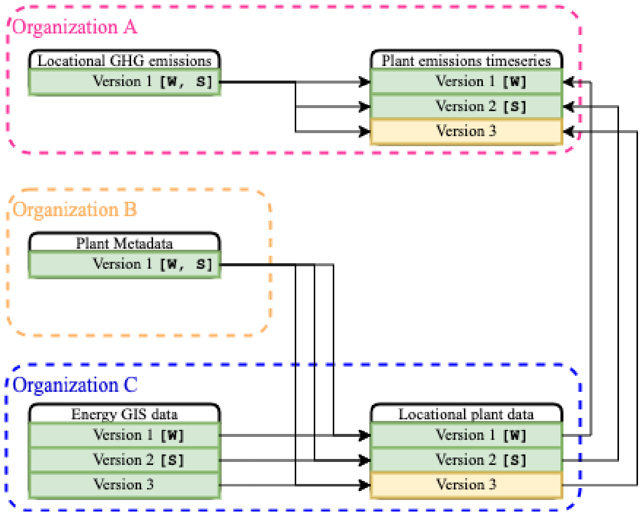

Fig. 3: Dataset version dependency graph after user D requests materialization. Announced versions are shaded yellow and materialized versions are shaded green. Materialized versions are annotated with **W** for weak references held against them by user D and **S** for strong references.

(charging user D's quota for the retention). However, the physical assets produced by materializing version 1 of this dataset will be eligible for collection as soon as any organization-defined retention policies do not apply to upstream versions.

IV. Evaluation

We implement a Depot prototype using OpenJDK 11, Python 3.10, Jupyter 7, MySQL 8, and RabbitMQ 3.10. We use Erlang 25.0 for messaging, PyArrow 12.0 for loading/storing Apache Parquet files, and Spark 3.2 for batch processing. We use this prototype to evaluate the performance cost incurred by Depot facilitating data pipeline execution. To do so, we have crafted a benchmark application, referred to as the "Tree Benchmark" to measure Depot's performance in handling transformations across a range of complexities. We run the benchmark using Eucalyptus (v5) cloud instances. We deploy Depot using a 2 node cluster each with 24 2.6GHz processors and 72GB of memory.

The Tree Benchmark measures the latency of materializing a dataset version where none of the input datasets, except for the ultimate upstream dataset, have been pre-materialized for a family of compliant datasets. Each dataset in the family is identified by a depth k and encapsulates a constant-length sequence of 64-bit integers. Datasets with $k > 0$ are defined from a transformation that performs a horizontal vector sum over exactly two datasets with depth $k-1$. The special-case datasets with $k = 0$ serve as the initial data source for the tree and are pre-materialized.

In our evaluation, we create and pre-materialize text-file-based datasets of $k = 0$, which consist of 50,000,000 random 64-bit integers. Each derived dataset transforms two input datasets by performing an element-wise sum to produce a new tabular sequence of 50,000,000 64-bit integers. For datasets $1 \leq k \leq 5$, we measure the latency between requesting materialization of a version and the persistence of the version's result. In each run, none of the intermediate versions, except $k = 0$, are materialized, so the tree must be traversed, and parent versions materialized first, the latency of which is included in our end-to-end latency result.

In practice, Depot exploits the inherent parallelism present in graph-based execution models: a dataset with two inputs can have both input trees built entirely in parallel. There is an implicit synchronization point before the materialization of a dataset can begin: all of its inputs must be materialized, but otherwise, Depot may be performing computation of a

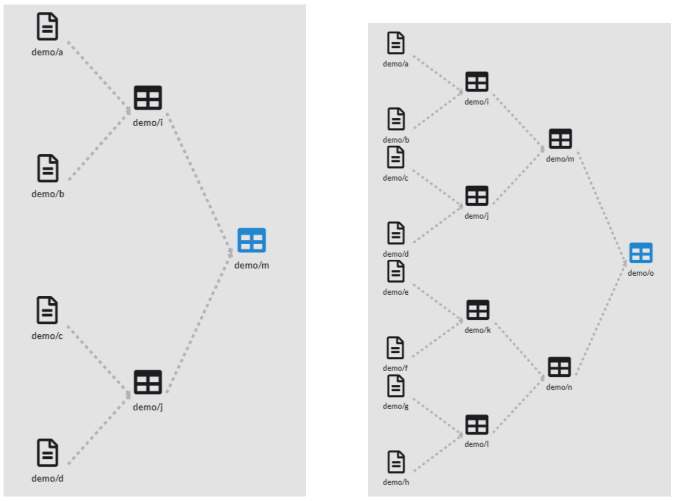

Fig. 4: Depot's dataset dependency graph visualizer depicting the Tree Benchmark datasets $k = 2$ (left) and $k = 3$ (right).

transformation for some datasets while scheduling announcements for others. This approach makes building a performance timeline and ascribing each time quantum to a distinct phase in Depot's execution model (e.g., computation, messaging overhead, IO) difficult, so we introduce explicit synchronization points between transformation computation, messaging, scheduling, and the data translation layer for this evaluation.

For example, when materializing a $k = 3$ dataset, Depot will schedule all transformations for the upstream $k = 1$ datasets, wait until all transformations are ready for execution, start transforming the $k = 1$ datasets, then wait for all transformations to complete before proceeding to the next phase for the $k = 2$ datasets. This synchronization (which we use only for evaluation purposes) makes estimates of end-to-end latency conservative because, by breaking the execution model up into synchronized phases, the latency of each phase is controlled by the maximum latency of each of its constituent operations. This break-up allows for a more transparent overall analysis of system-imposed overhead by eliminating overlap between phases. Because we have eliminated the overlap, the following results represent an upper-bound on the latency a user would perceive.

Our end-to-end latency results are depicted in a stacked fashion in Fig. 5. The latency of the compute phase increases linearly with k because each additional layer in the dataset tree requires an additional set of transformations, which can all be executed in parallel. The translate phase has a relatively high baseline latency, which then increases much more slowly. This is because the initial translation for $k = 0$ involves extracting integers from an unstructured text file. However, each additional layer for datasets $k > 0$ can use Depot's optimized table persistence and loading mechanisms. Finally, the delays due to signaling and scheduling within Depot are negligible compared to computation/IO work and also scale linearly with the depth of the dataset tree.

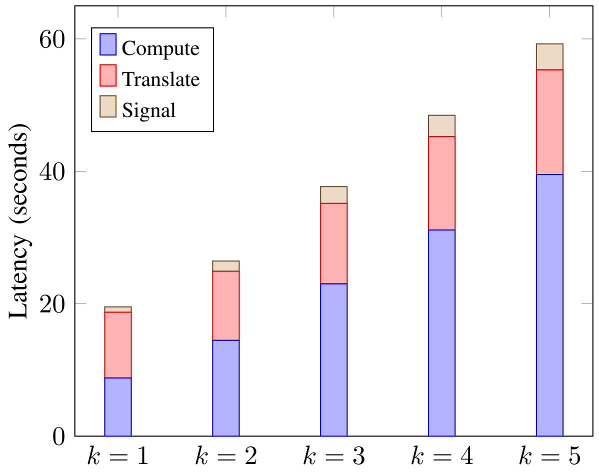

Fig. 5: End-to-end depot transformation latency in seconds for dataset dependency depth k (x-axis). Materialization for depth k triggers materialization of dependent datasets at depth $k - 1$. The performance is an upper bound on the latency a user perceives because we prevent materialization overlap for clarity of evaluation.

A. *Evaluating Depot with Streaming Data*

To incorporate a streaming capability, we extend Depot using Apache Kafka [9] (v3.5). Kafka is a popular and scalable, distributed event streaming platform that implements topic-based publish/subscribe pattern. Figure 6 shows the Depot streaming architecture in the prototype. Depot streaming allows datasets to be announced or materialized in response to "events" observed in a Kafka stream (denoted by a Kafka topic). It does so via a consumer agent that subscribes to topics of interest. The agent triggers Depot actions in response to data consumed from the topic. Note that announced datasets derived from a stream source require the stream data to be preserved until the dataset is eventually materialized. Thus, our current prototype uses the maximum duration (globally, for the Depot deployment) for announcements of stream sources. Supporting fine grain (per topic) preservation is the subject of ongoing work.

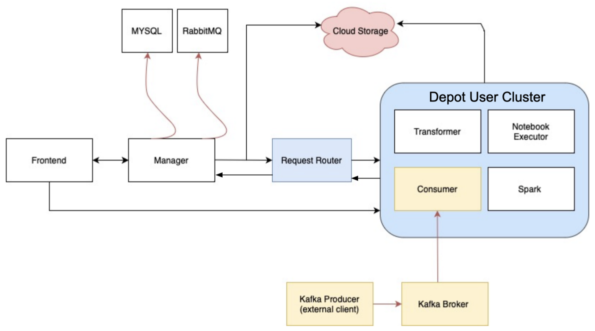

Fig. 6: Depot Stream Processing Architecture

We evaluate the performance of Depot stream processing using a digital agriculture application from the UCSB SmartFarm project [20]. It is an environmental monitoring application deployed at the Lindcove Research Extension Center [27] located in Exeter, California. The application continuously gathers and processes "nanoclimate" data from a large test facility for growing citrus [26]. The application collects and publishes a set of climate measurements to Depot from the facility every 10 minutes.

We use the application to measure the performance of stream materialization and processing in Depot. We present a breakdown of the average latencies (over 10 executions of the application) in seconds in Table I. Materialization consists of accessing Kafka (row 1), creating a dataset in Depot and storing it in cloud storage (row 2), invoking the user's transformation script (row 3), and execution the script via PySpark (row 4). The script simply converts the Kafka data to a Spark dataframe and exits; it is at this point that user code would manipulate the data as desired using distributed Spark workers. Row 5 shows the total latency (18.4 seconds) when using a single-machine deployment for the services (with no spark workers or network delay). Executing the same operations in Depot results in a total latency of 19.8s, indicating that Depot and/or distributed deployment results in an overhead of 7.6%.

TABLE I: Average latency for stream materialization

Stream monitoring	0.03s
Creating dataset	2.7s
Script invocation	3.4s
Script execution	12.3s
Total (single machine)	18.4s
Depot end-to-end	19.9s

While this performance profile acceptable for an application with a 10-minute duty cycle, we are pursuing additional optimization opportunities to ensure low overhead. Specifically, we are exploring alternatives that will allow users consuming a stream to allocate resources from their quotas explicitly to avoid startup overhead (e.g. Spark workers introduce an additional 10s) via pooling and by keeping services "warm" to avoid cold start overhead, as part of our future work.

V. Conclusion

Community data curation is a vital model of collaboration stimulating research through data sharing. We have presented Depot, a new data curation system that builds upon and extends the data lake data management model. Specifically, Depot adds community-aware data management features that decouple data origination from consumption and that simplify and facilitate data sharing and exploration, policy enforcement, and resource accounting and consumer-driven attribution. Key to the Depot design is the lazy materialization of datasets which reduces computation and storage costs until the resulting dataset is needed by the consumer. The evaluation of our prototype implementation demonstrates that this new model is viable in practice.

We see many opportunities for future research in this area. First, we plan to develop mechanisms for sharing the storage costs among data consumers with strong references to datasets and eliminating up-front quota requirements. Furthermore, we are working on fine-grained access policies to foster more community sharing. Finally, we will investigate new rights-transfer policies that are effective and efficient with this combination of mechanisms.

References

[1] The astronomer's telegram, 2023. https://astronomerstelegram.org.

[2] S. Benjelloun, M. El Aissi, Y. Lakhrissi, and S. El Haj Ben Ali. Data lake architecture for smart fish farming data-driven strategy. *Applied System Innovation*, 6(1), 2023.

[3] A. Chien. Right Place, Right Time Carbon Emissions Service, 2023. http://ripit.uchicago.edu, [Online; accessed 27-July-2023].

[4] Citisci on-line venue, 2023. https://www.citsci.org.

[5] Elasticsearch, 2018. https://www.elastic.co [Online; acc. 06-June-2023].

[6] M. Wilkinson et al. The fair guiding principles for scientific data management and stewardship. *Scientific Data*, 3(160018), 2016.

[7] T. Kluyver et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *Positioning & Power in Academic Publishing: Players, Agents and Agendas*, volume 2016. IOS Press, 2016.

[8] The eucalyptus open source cloud, 2023. https://www.eucalyptus.cloud.

[9] Nishant Garg. *Apache kafka*. Packt Publishing Birmingham, UK, 2013.

[10] The gamma-ray coordinates network, 2023. https://gcn.gsfc.nasa.gov.

[11] Globe observer on-line venue, 2023. https://observer.globe.gov.

[12] R. Hai, S. Geisler, and C. Quix. Constance: An intelligent data lake system. In *International Conference on Management of Data*, 2016.

[13] inaturalist on-line venue, 2023. https://www.inaturalist.org.

[14] W. Inmon. The data warehouse and data mining. *CACM*, 39(11), 1996.

[15] W. Inmon. *Data Lake Architecture: Designing the Data Lake and Avoiding the Garbage Dump*. Technics Pub., 2016.

[16] HA. Jacobsen. Topic-based Publish/Subscribe. In *Encyclopedia of Database Systems*. Springer, 2009.

[17] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101, 2014.

[18] P. Khine and Z. Wang. Data lake: a new ideology in big data era. In *ITM web of conferences*, volume 17, 2018.

[19] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. Baveja. Lambda architecture for cost-effective batch and speed big data processing. In *IEEE international conference on big data*, 2015.

[20] C. Krintz, R. Wolski, N. Golubovic, B. Lampel, V. Kulkarni, B. Sethuramasamyraja, B. Roberts, and B. Liu. SmartFarm: Improving Agriculture Sustainability Using Modern Information Technology. In *KDD Workshop on Data Science for Food, Energy, and Water*, August 2016.

[21] Nature's notebook, 2023. https://www.usanpn.org/natures_notebook.

[22] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2009.

[23] D. Sarramia, A. Claude, F. Ogereau, J. Mezhoud, and G. Mailhot. Ceba: A data lake for data sharing and environmental monitoring. *Sensors*, 22(7), 2022.

[24] R. Simpson, K. Page, and D. De Roure. Zooniverse: observing the world's largest citizen science platform. In *International conference on world wide web*, pages 1049–1054, 2014.

[25] Diomidis Spinellis. Git. *IEEE software*, 29(3):100–101, 2012.

[26] Citrus Under Protective Screening, 2023. https://citrusindustry.net/2017/08/18/cups-test-begin-lindcove-rec/ [Online; accessed 14-July-2023].

[27] UCANR Lindcove Research and Extension Center, 2023. http://lrec.ucanr.edu [Online; accessed 14-Feb-2023].

[28] Zooniverse on-line venue, 2023. https://www.zooniverse.org.