Git is for Data

Yucheng Low*
ylow@xethub.com
XetData Inc.
Seattle, WA, USA

Rajat Arya*
rajat@xethub.com
XetData Inc.
Seattle, WA, USA

Ajit Banerjee*
ajit@xethub.com
XetData Inc.
Seattle, WA, USA

Ann Huang
ann@xethub.com
XetData Inc.
Seattle, WA, USA

Brian Ronan
brian@xethub.com
XetData Inc.
Seattle, WA, USA

Hoyt Koepke
hoytak@xethub.com
XetData Inc.
Seattle, WA, USA

Joseph Godlewski
jgodlewski@xethub.com
XetData Inc.
Seattle, WA, USA

Zach Nation
zach@xethub.com
XetData Inc.
Seattle, WA, USA

ABSTRACT

Dataset management is one of the greatest challenges to the application of machine learning (ML) in industry. Although scaling and performance have often been highlighted as the significant ML challenges, development teams are bogged down by the contradictory requirements of supporting fast and flexible data iteration while maintaining stability, provenance, and reproducibility. For example, blobstores are used to store datasets for maximum flexibility, but their unmanaged access patterns limit reproducibility. Many ML pipeline solutions to ensure reproducibility have been devised, but all introduce a degree of friction and reduce flexibility.

In this paper, we propose that the solution to the dataset management challenges is simple and apparent: **Git**. As a source control system, as well as an ecosystem of collaboration and developer tooling, Git has enabled the field of DevOps to provide both speed of iteration and reproducibility to source code. Git is not only already familiar to developers, but is also integrated in existing pipelines, facilitating adoption. However, as we (and others) demonstrate, Git, as designed today, does not scale to the needs of ML dataset management. In this paper, we propose XetHub; a system that retains the Git user experience and ecosystem, but can scale to support large datasets. In particular, we demonstrate that XetHub can support Git repositories at the TB scale and beyond. By extending Git to support large-scale data, and building upon a DevOps ecosystem that already exists for source code, we create a new user experience that is both familiar to existing practitioners and truly addresses their needs.

1 INTRODUCTION

Machine Learning (ML) is a data-driven field where data quality can have a greater impact on results than modeling innovations [1]. In academic settings, high quality *golden datasets* are held constant to isolate and compare model improvements. Conversely, in industry settings, iterative steps such as data collection, analysis, labeling, modeling, and deployment occur concurrently. For example, an update to the data processing pipeline will create new data, which will trigger modeling algorithmic updates and require a new model to be built and deployed. In this world, **code and data co-evolve**.

Despite their codependency, code and data are historically managed by separate systems. Solutions that track both code and data often require heavy integration with MLOps ecosystems, so many teams opt for lightweight solutions (e.g., Slack or Google Docs) to synchronize communication as needed. These unreliable channels become a liability when tracing the provenance of a specific change.

With code and data changing simultaneously, identifying the root cause of model performance improvements or degradations becomes difficult. Even simple tasks—like evaluating whether a new model developed by another team has improved performance—are challenging, requiring all teams to align on data, labels, and input/output processing methods.

In this paper, we focus on the problem of **dataset management** as the core challenge in industry MLOps [2]. We contend that data engineering is a software engineering discipline and should be addressed with the same set of tools. By treating MLOps as an extension of DevOps with the problem of scale, we demonstrate that Git, with appropriate enhancements, can provide a complete solution that will satisfy most dataset management requirements.

In Sec. 2, we first provide an overview of industry ML dataset properties and explain their usage in Sec. 3. Next, Sec. 4 poses a thought experiment on how to use Git to manage a tiny dataset. Sec. 5 covers the benefits of Git and leads into Sec. 6, where we show how we scaled Git to handle TB-scale real-world ML datasets with our system called **XetHub**.

2 DATASET

A **dataset** is an organized collection of data comprised of anything from tables and free-text to images and videos, usually composed to achieve a particular objective. ML datasets are frequently created with the goal of meeting an ML objective (e.g., training a speech-to-text model) and often require data labeling by human annotators. **Industry ML datasets** can be further distinguished by these common characteristics and requirements:

*Joint first author

 13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.

Infrequent Updates While raw data is always changing, industry ML datasets often require human labeling and quality control to ensure accuracy: a costly and inherently slow batch process that can take weeks. As such, datasets suitable for ML training tasks are generally infrequently updated.

This is convenient, as tracing back changes in model performance requires holding either the data or code constant; frequently changing datasets are disruptive to model evaluation. These datasets thus do not benefit from being updated more frequently than the latency of label acquisition, model development, and model evaluation. From the authors' experiences with ML teams, *monthly* updates are typical, aligning with sprint schedules which allow new model architectures to be evaluated on a regular cadence.

While streaming data and online learning regimes for MLOps exist, they demand significantly more sophistication in both modeling and deployment to monitor real-time model performance and avoid model drift [3]. These settings are comparatively rare; the majority of industry ML are in the batch regime.

Small Changes, Many Copies Dataset updates are often incremental: appending new data, evicting expired data, or updating labels. From each version, engineers and scientists create data subsets for rapid model iteration, data exploration, and other experimental uses. The subsets may then be reused or pooled by other teams for "off-label" usage, resulting in many copies of similar data within a company.

Arbitrary File Types In industry, datasets are comprised of arbitrary file types as dictated by the tools used by the team and broader organization. For instance, log or sensor data could be simple text files or proprietary serialized formats, depending on the recording equipment. Similarly, every ML library (e.g., TensorFlow or PyTorch) consumes and produces different file formats.

File formats are also defined by performance requirements. Large tensor files may be sharded for more efficient distributed training, and datasets with a large number of small files are often compressed to reduce the number of files and increase I/O performance.

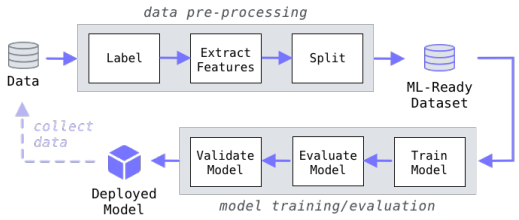
While there are many possible data formats, the number of file types used within a single domain is quite homogeneous. Every genomics lab will have FASTQ files, and every team using TensorFlow will use tfrecord files.

Flexible Data Organization Industry ML teams prefer to organize their datasets in whatever file layout is simplest for them, without the need to restructure or reorganize data to fit a specified schema. Imposing rigid structures on organization adds friction to existing workflows and distracts from ML innovation.

Contains Code Since file types and schemas are closely tied to models and tools, which can both change as requirements evolve, code that reads or generates data must evolve alongside the data itself. Preprocessing rules and augmentation methods are also closely bound with the data and cannot be versioned independently. Therefore, code is a fundamental part of datasets and cannot be managed separately.

3 ML PIPELINES

A typical dataset goes through a ML pipeline with steps like data preprocessing (e.g., clean outliers and add structure), feature extraction (e.g., dimensional reduction), modeling (e.g., training and evaluation) and model deployment. Intermediate results may be computed on the fly or stored for performance reasons. For instance, an ML based feature extraction model may be too slow to compute on the fly, while the fast image augmentation methods (e.g., random rotations) can be computed as needed. Intermediate pipeline outputs are sometimes useful for multiple teams; a BERT-embedded [4] text dataset may be useful for both spam classification and sentiment analysis.



The final result of a successful pipeline run is a deployed **ML model**. While this paper does not address model artifact management and deployment challenges, we note that the model may itself be consumed by a different pipeline. For instance, the model delivered by one team may be used as a feature extractor for another team.

3.1 PROVENANCE AND REPRODUCIBILITY

As the dataset is transformed through a pipeline, it is frequently necessary to track the **provenance** of the data: how it was produced, and where it came from.

Record-level provenance is useful for identifying bad records. In the process of training an ML model, users may catch problematic records and want to diagnose the issue by back-tracking to identify the original input. Record-level provenance is frequently addressed by referring to unique IDs associated with each row of data. Sophisticated solutions here are likely not worthwhile due to the breadth of tooling and data manipulation methods involved.

Alternatively, **dataset-level provenance** is useful for the understanding and debugging of ML pipelines. To identify the cause of a regression in model performance, scientists need to know the stages involved in model production. To accurately attribute model performance improvements over time, scientists need to track the changes in both code and data over time.

Dataset-level provenance provides reproducibility: the ability for *other people in the organization* to inspect the code and data which goes into producing a model, and to reliably repeat a pipeline to obtain the same outcome. The ability to trace the evolution of a model—including the code and data that went into it—is necessary for an ML team to understand how to deliver improved performance. Certain regulated industries may also require reproducibility for compliance reasons.

Industry applications of ML commonly stitch together complex pipelines [5], reusing data and ML libraries from other teams. The loss of data provenance through these cross-team, cross-system workflows result in reproducibility challenges [6] that negatively impact model performance in production.

To support the co-evolution of data and code, we believe that provenance systems should be light-weight, integrating seamlessly into existing source code management systems. **Data Dependency** and **Code Dependency** are fundamentally the same problem and solutions must address both identically.

3.2 FLEXIBILITY

Just as there is no single source code management pattern that works for everyone, we believe that there is no single ML pipeline pattern that is universally optimal. We have repeatedly observed a pattern where every team architects their own pipeline system, finding existing solutions too prescriptive or insufficient for their requirements.

While strict pipeline systems provide reproducibility and the ability to *conduct science*, business decisions take priority and sometimes it is necessary to *"fix it in production"*. For instance: to rapidly revoke an output label to eliminate model bias [7].

As such, we believe that data management systems should be *unopinionated*, integrate with existing tools quickly and flexibly, and enable any and all workflows.

4 THOUGHT EXPERIMENT: 1MB DATA

We believe that the challenges of dataset managment is *"merely"* that of scale and size. That is, many MLOps challenges reduce to DevOps if sufficient scalability is provided. To motivate this belief, consider a thought experiment in this limiting case: What if the size of all collected data fits in less than 1MB?

Using the workflow of a team working on ML models for a Personal Assistant (PA) such as Google Assistant as a running example, we make the unrealistic assumption that all the data fits in 1MB and is thus stored and accessed through **Git** [8].

To build the PA, the team collects 2 datasets: a speech database comprising of audio samples and annotated text, and a database mapping text queries to intent. The data is held in a transactional database with additional fields for managing acquisition and labeling efforts (e.g., speaker profiles to ensure data diversity, contradictory labels for manual merging).

Once a month, snapshots of both audio and intent databases are collected. Since the snapshots are < 1MB, they are checked into a Git repository and tagged. If export bugs are encountered, a new version is committed with updated tags. The raw dataset has **many file types** comprising of WAV files for audio samples, text files with transcriptions, and JSON records for intent structures. This data is partitioned into train/validation/test folders to ensure common dataset splits, and stored alongside Python code for reading WAV files and a library to parse the intent schema. When data layout changes occur (e.g., to add new intents), the raw files and parsing code are updated together. All historical versions are preserved during this **co-evolution of code and data**.

Throughout the month, ML engineers iterate to optimize model performance. Each engineer checks out the model repository, which tracks the dataset repository as a dependency, not unlike any other source code dependency. Integrated build tools or libraries may be used to manage this dependency. The engineers primarily work on the modeling code, and use small data subsets to validate correctness. As they experiment, they check in additional code for running filters and producing tfrecord files, using branches to easily collaborate with teammates on both modeling and data processing code. Pull requests are used for review, updating the main branch with their code and data changes on approval. **Continuous integration** is used to generate the final model artifact, which is then deployed. Post-deployment, Git is used as an artifact repository for inference code to be stored together with the model.

The use of a Git repository benefits not just the team working on the PA. An adjacent team working on a new end-to-end speech-to-intent model can fork the entire repository for their own experiments. Other teams experimenting on text-to-speech may clone the model artifact repository, easily accessing the embedded inference code.

Finally, the **provenance** provided by the Git repository enables effective regression testing. Scientists can use the repository commit history to compare differences between the previous deployed model and the current deployed model to identify if the regression is due to a change in code or data. Fixes to either code or data can be submitted via a pull request to update the deployed model.

5 IF GIT WERE SCALABLE

As datasets today are typically larger than 1MB, scalable storage solutions to 1TB and beyond are needed. Git performs poorly with large files and Git's single server endpoint is a bottleneck. Users typically offload datasets to blobstores, where proper versioning and history tracking require some maintenance.

However, if Git were scalable to the repository sizes required, it could provide the flexibility and capability to serve the needs of industry ML. Solutions such as Git LFS [9] and DVC [10] provide a light-weight facade for adding large files to Git repositories but do not provide sufficient integration to support the needs of industry ML datasets as described in Sec. 2.

To scale Git, we approach Git differently—as more than a system or a service for source code version control. At a low level, Git serves as a protocol and a metadata store for a filesystem with coarse grained atomic writes; at a high level, Git is an entire ecosystem of tools and services.

Git is a Database Git internal storage is a well-optimized content-addressed object database for small objects. Significant optimizations have been made so that clients can download the minimum information necessary to construct commits and subdirectories [11, 12]. We have benchmarked Git with rudimentary tuning to over $550M$ 128 byte objects and we believe that, with more tuning [13], Git can scale to billions of objects. The Git object database also has many opportunities for further optimizations [14].

Git is a Filesystem Metadata Store Git provides a directory tree layout over blob contents. The sparse-checkout, shallow clone, and sparse-index features in every Git client provide an efficient way to obtain the metadata required to fully describe a partial or complete directory structure of a repository for any given version.

Git is a Protocol Git is merely a backend choice for representing versioned filesystem metadata. Frontends can be architected to speak the Git protocol without any requirement to use Git itself.

Git is for Collaboration Git/GitHub have defined common patterns around developer collaboration: issues, branching, forking, and pull requests. These patterns have helped establish modern DevOps practices and cultures around testing, automation, and deployment. With the development of Infrastructure as Code [15, 16], version control has become the de-facto way to collaborate on all aspects of DevOps from development through deployment.

We aim to extend the benefits of Git to both code and data, with a user experience that seamlessly supports ML datasets and arbitrarily large files.

6 SCALING GIT TO DATASET MANAGEMENT

To solve the industry ML dataset problem, we realized that providing users with a Git-like dataset management does not provide a good user experience. With the broad familiarity of Git among users, as well as the extensive collection of developer tools surrounding Git, it is insufficient to merely look like Git; we must embrace Git in its entirety. To that end, we propose a system design called XetHub to enable Git at scale.

The XetHub design factors the dataset representation problem into two parts: filesystem metadata and data.

6.1 METADATA

Git is an efficient store of versioned filesystem metadata, scaling comfortably to hundreds of millions and potentially billions of small objects. We make use of Git to store pointer representations to the actual data. While here the design appears similar to Git LFS or DVC, the key difference is that we take over data storage responsibility for all data, which allows us to transparently manage and scale data storage.

We built this system entirely on the the Git Clean/Smudge Filter protocol mechanism, using a broad wildcard to capture every file. Since every file passes through our filter, we are able to optimize the data storage locations for each file.

6.2 DATA

Our data stack makes use of a data deduplication method backed by a Content-Addressed Store (CAS). The data deduplication method is designed to leverage large objects (which are more efficient to store, communicate and manage), while maintaining small block sizes which improve dedupe performance.

6.2.1 Content Defined Chunking. The Content Defined Chunking procedure is a data chunking procedure that generates variable-sized chunks which are *data-dependent*. By determining chunk boundaries based on the the contents of the data, the chunking procedure can be robust to both data insertions and deletions. The typical procedure involves scanning the data stream with a rolling hash algorithm [17, 18], and generating a chunk boundary when the hash meets a particular criterion. For instance, the simple criterion $c(hash) = hash \bmod 2^{12} == 0$ will, assuming random file contents, generate chunks averaging $2^{12} = 4096$ bytes.

We implemented several of the optimizations described by the FastCDC algorithm [19] to accelerate the chunking process but used a different method to provide normalized (more Gaussian distributed) chunk sizes: assuming uniformly random hash values, the chunk size is described by the geometric distribution $X\ Geo(p)$ where p is the probability the rolling hash value meets the criterion. (For instance, $p = 2^{-12}$ in the example above). This means that using the rolling hash procedure naively will result in a large number of tiny chunks which increase the overhead required to represent deduped data. See Fig. 2 for an example.

One of the optimizations in the FastCDC paper is a normalized chunking procedure where an adaptive criterion is used to approximately normalize the chunk sizes, producing a chunk size distribution which is a mixture of two geometric distributions.

We propose an alternative we call **Low Variance Chunking** where we simply sequentially perform k runs of rolling hashes with different hash seeds, scaling the criterion appropriately to achieve the same target chunk size. For instance in our system, we use $k = 4$ different rolling hashes with each hash targeting a chunk size of 4096 bytes. The resultant chunks have an average of $16KB$ with sizes distributed according to $NB(k, 2^{-12})$ which for sufficiently large k is approximately Gaussian.

In Fig. 2 we plot the chunk distributions provided by low variance chunking compared with standard chunking. We observe that the low variance chunking procedure provides more Gaussian distributed chunk sizes, and avoids both tiny chunks and massive chunks. We believe this low variance procedure provides a simpler parameterization for balancing dedupe quality and overhead.

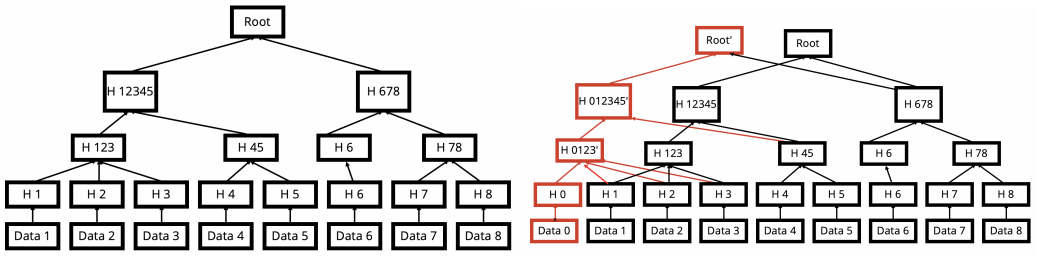
6.2.2 Data Deduplication. The core datastructure underlying the data deduplication method is a Content-Defined Merkle Tree (CDMT) [20]. Unlike a typical Merkle Tree with a fixed branching factor, each node in a CDMT has a variable number of children (Fig. 1a). The tree is built with a method similar to Content Defined Chunking, and allow modifications without large changes to the tree structure.

Firstly, files are chunked with the chunking procedure described in Sec. 6.2.1. A Merkle Tree is built up by grouping chunk hashes together, also using Content-Defined Chunking procedure: a hash is on a chunk boundary if the hash meets criterion $hash \bmod 4 == 0$. We also enforce that chunks must comprise of between 2 and 8 hashes. Each chunk of hashes are then merged into a single parent node. The procedure is repeated until there is only a single root node. The hash value at the root node is called the **MerkleHash** of the file.

Due to the use of the Content-Defined Chunking procedure during tree construction, the CDMT datastructure allows for insertions, deletions, and modifications to be performed without substantial changes to the tree, allowing most of the tree to be preserved across chunks (Fig. 1b) thus minimizing overhead for data changes.

Finally, since the MerkleDAG (collection of Trees) is simply represented as a set of hashes, it is trivially a Grow-Only Set CRDT [21], allowing for simultaneuous conflict-free modifications. We maintain the MerkleDAG on a per-repository basis, storing incremental changes in Git notes [22]. The MerkleDAG hence covers all branches, allowing data dedupe capability to automatically span all branches. Similarly, forks or clones of the repository automatically inherit the same MerkleDAG and all dedupe capabilities.

6.2.3 Data Storage. The final piece of the system is the actual data storage for the deduplicated chunks. We rely on a Content



(a) Merkle Tree (b) Merkle Tree Insertion

Figure 1: (a) Content Defined Merkle Tree. Each leaf chunk is derived from a Content Defined Chunking procedure. Nodes are merged using a simple hashing rule: a partition is inserted whenever the hash modulo a target child count equals 0 with constraints on a minimum / maximum child count. (b) Insertion of a new chunk (Data 0) can maintain tree stability. New nodes are in red.

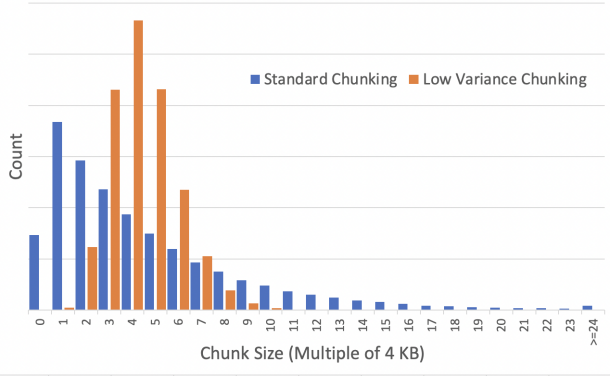


Figure 2: Example chunk size distribution using a standard rolling hash chunking procedure, vs the low variance chunking procedure on 4.3GB of compressed CentOS disk images. We observe the low variance chunking procedure is more Gaussian distributed and avoids both excessively small and large chunks.

Addressed Store (CAS) where the key used to retrieve a piece of data, is a hash function of the data.

A typical way to use a CAS to store deduplicated chunks is to simply store each chunk as its own key. While small chunk sizes have better deduplication performance, this creates small objects which lead to large key storage overheads in the CAS, as well as large network overheads for uploading and downloading objects. We will ideally like to have small dedupe chunk sizes, but large CAS entries.

A key observation is that we can use the CDMT to convert between files, chunks and CAS entries.

When new files are added to the system, they are first chunked with the method described in Sec. 6.2.1, with a CDMT constructed as described in Sec. 6.2.2. Any new chunks which do not exist in any other versions of the repository are directly concatenated to a maximum of 16MB, and stored stored as a single object in the CAS. A CDMT for this new object is also derived and stored.

The system hence comprises of two groups of CDMTs. The first group of CDMTs have roots which are files, and the second group of CDMTs have roots which are CAS entries. Both groups of CDMTs share chunks: by construction, every leaf chunk must be both a leaf of some file root and a leaf of some CAS entry root. The use of the Content Defined Chunking procedure also mean both groups of CDMTs share many interior nodes, reducing representation overhead.

To reconstruct any file root, we simply intersect the file CDMT with the CAS CDMTs to resolve the set of CAS object ranges needed to reconstruct the file (Fig. 3).

This deduplication strategy provides the best of both worlds:

- Effective data dedupe: Data dedupe with small chunks and support for both insertions and deletions.
- Low CAS overhead: Large CAS object sizes result in low storage and communication overhead.
- High data locality: If a range in a CAS object is required, it is likely that the rest of the CAS object is also required.

6.2.4 Data Dependent Chunking. While the default chunking strategy provides an excellent baseline for many datasets, specialized chunkers can be provided for special file types. For instance: CSV files can be preferentially chunked on line breaks and Numpy arrays can be preferentially chunked by the first index stride length. This can improve dedupe performance significantly for those file types enabling free subsampling and reordering: both common dataset operations. We explore this further with the benchmarks in Sec. 6.4.

6.3 GIT INTEGRATION

As the entire metadata stack is exactly Git, we are able to support all Git interactions transparently without requiring the user to learn any additional commands, retaining compatibility with existing tools in the Git ecosystem. See Table 1 for a comparison of how command line interfaces differ between Git LFS, DVC, and Git Xet.

While the merge concept in Git does not extend well to large binary files, a merge conflict informs the user when an unexpected simultaneous modification by another user has occurred. Git also provides the user the opportunity to resolve it manually, or by force (overwrite). This is unlike typical object store workflows where conflicting changes can be overwritten without notification.

While the Git Filter protocol is quite efficient, deeper integration with Git will bring improved performance. Alternatively, in Sec. 7 we consider the use of user-mode filesystems as the more scalable solution for user interaction.

6.4 CORD19 BENCHMARK

We benchmark our dedupe performance using the CORD-19 dataset [23], a text corpus of academic papers about COVID-19 containing

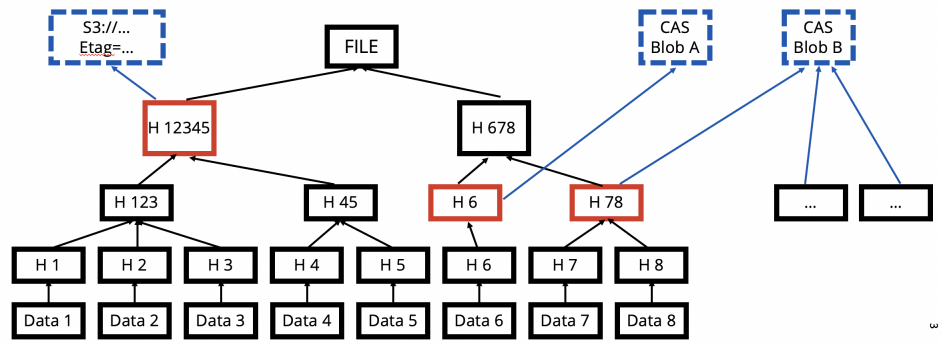


Figure 3: To reconstruct a given file, the Merkle Tree is traversed to find data sources to concatenate. The solid red nodes indicate the set of nodes which are sub-ranges in other storage sources (dashed blue nodes). `H12345` can be read from an existing S3 object, `H6` can be directly queried from the CAS, and `H78` is a sub-range in another CAS entry (Blob B).

full text paper contents, authors, and abstracts. Document embeddings are also provided. This dataset is an example of a real-world growing dataset, with regular new releases of papers and a complete history of prior versions. We evaluate the cost of using Git with either LFS or Xet integration for dataset version control, using the last 50 versions of the CORD-19 dataset spanning from 2021-02-01 to 2022-06-02. We do not test other Git large file solutions such as DVC and Git Annex [24] as they all only perform file-level deduplication and so will have identical storage characteristics as Git LFS.

Each version of the dataset comprises of small JSON files (up to 700k in the final dataset) with paper information, as well as 2 large CSV files of metadata and document embeddings respectively. We evaluate Xet with 2 different chunkers: the generic all-purposes chunker (Labelled Xet) and Xet with a CSV aware chunker (Labelled Xet+CSV).

The combined uncompressed size of storing all 50 versions is 2.45TB. Each version was extracted, added, and checked into Git. For Git LFS, LFS is configured to track the large CSV files (metadata and embeddings). Xet is used in its default configuration. We show in Fig. 4 that 2.45TB of raw data can be stored with 545GB (4.6x reduction) with rudimentary separation of data and metadata using Git LFS. Xet can reduce this further to 287GB (8.7x reduction), and Xet+CSV can reduce the storage requirements to merely 87GB, or a 28.8x reduction. These measured sizes include all Git repository information and LFS/Xet metadata. Notably, for Xet+CSV, the size of the final version of the dataset is 81.5GB; only 5.5GB is needed to maintain all previous versions.

Note that data compression, which would further reduce both disk space and network utilization, is not used in this benchmark.

Next, in Fig. 5 we test the incremental cost of adding a branch storing a dataset split. Both the metadata file and the embedding file were split into 75% – 25% parts. In the *random split* strategy, the rows are randomly shuffled into two files. In the *aligned split* strategy, the file is simply partitioned. LFS required complete storage of all split files requiring 16GB of storage. Xet dedupes the random split to 11GB, while Xet+CSV requires only 1.6GB, or a 10x improvement over the baseline. The aligned split is easy for both chunking strategies, requiring 185KB for Xet and 173KB for Xet+CSV. The Xet+CSV failed to dedupe every row in the random split as short CSV rows may get merged to maintain reasonable chunk sizes.

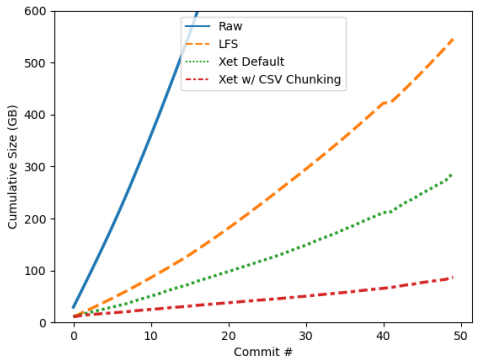


Figure 4: Cumulative storage cost for committing 50 CORD-19 dataset versions from 2021-02-01 to 2022-06-02. The Y axis is truncated for readability. LFS required 545GB, Xet required 287GB including a 2.4GB MerkleTree. Xet+CSV required 87GB including a 1.7GB MerkleTree. See Sec. 6.4 for details.

7 BEYOND GIT CLONE

As we scale Git to large repositories comprising of TBs of history and millions of files, the typical Git user experience of cloning an entire repository becomes a performance bottleneck. It should not be required to clone the entire repository to explore a dataset, or to access a few files.

7.1 USER-MODE FILESYSTEMS

To alleviate this issue we provide a `mount` command (Table 1) which exposes a Xet repository as a file system folder. The filesystem view of a dataset provides transparent delayed materialization of the repository, allowing all users to get a single common view of a repository efficiently.

Users can gain access to large datasets immediately and freely use local tools to explore datasets on their own machines. Images and audio can be directly loaded with native applications. Random access file formats such as Zip files allow partial contents to be extracted without a complete download. Parquet files and Sqlite databases can be directly queried with familiar native tools. This

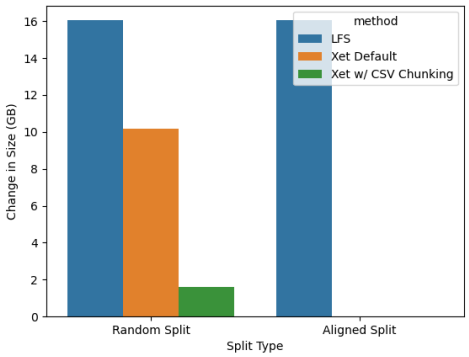


Figure 5: Incremental storage cost of adding a branch with a dataset split. In the *random split*, the rows are randomly shuffled into two files. In the *aligned split*, the file is simply partitioned. LFS required complete storage of 16GB. Xet dedupes the random split to 11GB, and 185KB for the aligned split. Xet+CSV requires 1.6GB for the random split and 173KB for the aligned split. Aligned split bars are too small to display See Sec. 6.4 for details.

Git LFS	Clone	`git clone [repo]`
	Add	`git lfs track [file]` `git add [file]`
	Push	`git push`
DVC	Clone	`git clone [repo]` `dvc pull`
	Add	`dvc add [file]` `git add [file].dvc`
	Push	`dvc push` `git push`
Git Xet	Clone	`git clone [repo]`
	Add	`git add [file]`
	Push	`git push`
	Mount	`git xet mount [repo]`
	Mount Branch	`git xet mount [repo] -r [branch]`

Table 1: Basic usage comparison. Git LFS [9] requires explicit decisions on what files/file patterns to store in LFS. Mistakes are complicated to resolve. DVC [10] exposes implementation and storage detail to the user and requires new commands for common operations. Git Xet integrates deeply and works transparently with all files and provides user-mode filesystem mount capabilities (Sec. 7.1)

is especially beneficial for data exploration, allowing developers to experiment on small data subsets easily.

The filesystem mount is also a natural way to support dataset dependencies in data engineering or ML training tasks as it allows tasks to simply read what they need without handcrafting a download manifest. Distributed training tasks are simplified as users do not have to address sharding and data distribution challenges. Training jobs are also accelerated as data can be streamed on demand avoiding large initial download costs which are hard to optimize.

Our user-mode filesystem implementation is built as a custom built local machine NFSv3 [25] server daemon, followed by an NFS mount from `localhost`. The key benefit of NFS vs FUSE (Linux Userspace Filesystem) is that NFS is widely supported and the protocol provides built-in semantics for the kernel to perform both metadata and data caching efficiently.

In Fig. 6 we demonstrate how with a local filesystem mount, useful queries for Parquet / SQLite databases can be performed using standard local tooling, but automatically fetching only a small fraction of the total size of the dataset.

Next, in Fig. 7, we evaluate the performance of the filesystem mount. and show that our uncached performance is comparable with the native DuckDB S3 connector for Parquet databases, though significantly slower for SQLite databases. However, once cached, we are comparable with local disk. Caching optimizations which are adaptive to file format is planned.

The file system view allow non cloud-aware tools to access large files and datasets with ease without requiring each tool to implement their own connector. A single common filesystem-based caching model further enables efficient repeated access for every tool without requiring each application to implement their own caching solution.

8 FRONTEND

The ability to store large datasets in Git raises the UX question of what an appropriate frontend for sharing and collaborating on datasets should look like. While GitHub and GitLab have demonstrated set of patterns for communicating source code history, diffs, and pull requests; appropriate patterns for datasets have no set precedence.

For instance, dataset cards [26] provide a way to document and communicate the current schema and metadata of a dataset, but not how the dataset has changed over time. Alternatively, while row-level diffs of structured files (CSVs, Parquet) can be displayed, this is unlikely to be useful when the diffs are large. Instead, appropriate use of sketch statistics and summarization could be used to visualize distribution changes to help quickly identify outliers, as well as data drift over time. This is a broad topic for future study.

Continuous integration as a pattern for source code quality extends naturally to datasets, providing a system for complex data quality analysis operations such as outlier detection, schema violations, etc.

9 RELATED WORK

Datahub [27] describes a Git-like repository model for structured data as well as a query language for versioned datasets. In this paper, we instead define a repository system for general unstructured data with system primitives (Sec. 7.1) that permit efficient querying of tabular data file formats such as Parquet/SQLite.

In [28], the authors similarly argue that "ML presents characteristics that are typical of software (e.g., it requires rich and new CI/CD pipelines), and of data (e.g., the need to track lineage)" and discuss the need for "queryable data abstractions, lineage-tracking and storage technology that can cover heterogenous, versioned, and durable data." We believe that the system described in this paper fulfills this need.

10 FUTURE WORK

The Git repository model is not suitable for capturing streaming data as each commit is a moderately costly operation. Pushing events

to a Git repository at high frequency is challenging. However, data can be directly streamed to Sec. 6.2.3 allowing only metadata to be batched and published to the Git repository at a coarser grained interval.

While the MerkleDAG is typically about 0.5% of the size of the repository, the MerkleDAG gets too large to download when the repository approaches the 10-100TB range. Improved ways to layout the MerkleDAG on disk, or to serve subgraphs of the MerkleDAG on demand are needed. Also, the MerkleDAG is append only and grows monotonically over time, which makes common industry use cases of data deletion and eviction (e.g., garbage collection, removed branches, legal deletion requirements) a challenge. Improving the MerkleDAG to support these capabilities is ongoing work.

The filesystem mount capability for Xet repositories enable "Time series" views of a dataset where multiple historical revisions or branches are simultaneously made available, allowing users to explore dataset changes over time. The dedupe capabilities improve the efficiency of such time-series views by automatically taking advantage of data commonalities across versions.

Next, the mount can also be used to capture fine-grained block level access, allowing **record-level provenance** to be inferred automatically without any additional tooling.

Finally, while the current mount implementation provides only read-only access to the repository, read-writeable mounts which maintain Git semantics will enable easy mutation of arbitrarily large data repositories.

11 CONCLUSIONS

At first glance, we integrate with Git in a comparable method as Git LFS. However, the core differentiation is the holistic set of tooling XetHub provides to fully support the needs of ML datasets by fully embracing the use of software engineering practices for data.

We believe that with the right architecture design, pre-existing systems for source control can be extended to fully support the dataset use case, addressing a significant fraction of dataset management needs while minimizing cognitive friction.

The significance is the observation that the needs around dataset management are not unique, and have been addressed by source code management tools. What is unique is only the scale at which it happens. By extending Git to support large-scale data, and building upon a DevOps ecosystem that already exists for source control, we create a new user experience that is both familar to existing practitioners and truly addresses their needs.

ACKNOWLEDGMENTS

We thank Carlos Guestrin and Shanku Niyogi for their helpful feedback.

REFERENCES

[1] Lucas Beyer, Olivier J. Hénaff, Alexander Kolesnikov, Xiaohua Zhai, and Aäron van den Oord. Are we done with ImageNet?, 2020. URL https://arxiv.org/abs/2006.07159.

[2] Pulkit Agrawal, Rajat Arya, Aanchal Bindal, Sandeep Bhatia, Anupriya Gagneja, Joseph Godlewski, Yucheng Low, Timothy Muss, Mudit Manu Paliwal, Sethu Raman, Vishrut Shah, Bochao Shen, Laura Sugden, Kaiyu Zhao, and Ming-Chuan Wu. Data platform for machine learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1803–1816, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450356435. doi: 10.1145/3299869.3314050. URL https://doi.org/10.1145/3299869.3314050.

[3] Jose G. Moreno-Torres, Troy Raeder, RocíO Alaiz-RodríGuez, Nitesh V. Chawla, and Francisco Herrera. A unifying view on dataset shift in classification. *Pattern Recogn.*, 45(1):521–530, jan 2012.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the North American Chapter of the Association for Computational Linguistics*, pages 4171–4186, June 2019.

[5] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems*, volume 28, 2015. URL https://proceedings.neurips.cc/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf.

[6] Sayash Kapoor and Arvind Narayanan. Leakage and the reproducibility crisis in ml-based science, 2022. URL https://arxiv.org/abs/2207.07048.

[7] Alex Hern. Google's solution to accidental algorithmic racism: ban gorillas. *The Guardian*, 2018. URL https://www.theguardian.com/technology/2018/jan/12/google-racism-ban-gorilla-black-people. Last Accessed: 2022-08-09.

[8] Scott Chacon and Ben Straub. *Pro Git*. Apress, 2nd edition, 2014. URL https://git-scm.com/book/en/v2.

[9] Git large file storage. https://git-lfs.github.com/, . Last Accessed: 2022-08-09.

[10] Data version control - dvc. https://dvc.org/, . Last Accessed: 2022-08-09.

[11] Derrick Stolee. Bring your monorepo down to size with sparse-checkout | The Github Blog. https://github.blog/2020-01-17-bring-your-monorepo-down-to-size-with-sparse-checkout/, 2020. Last Accessed: 2022-08-09.

[12] Derrick Stolee. Make your monorepo feel small with Git's sparse index | The Github Blog. https://github.blog/2021-11-10-make-your-monorepo-feel-small-with-gits-sparse-index/, 2021. Last Accessed: 2022-08-09.

[13] Taylor Blau. Scaling monorepo maintenance | The Github Blog. https://github.blog/2021-04-29-scaling-monorepo-maintenance/, 2021. Last Accessed: 2022-08-09.

[14] Derrick Stolee. Git's database internals i: packed object store. https://github.blog/2022-08-29-gits-database-internals-i-packed-object-store/, 2022. Last Accessed: 2022-08-29.

[15] Infrastructure as Code - Introduction to DevOps on AWS. https://docs.aws.amazon.com/whitepapers/latest/introduction-devops-aws/infrastructure-as-code.htmlb, 2020. Last Accessed: 2022-08-09.

[16] Pulumi - Universal Infrastructure as Code. https://www.pulumi.com/, 2022. Last Accessed: 2022-08-09.

[17] Andrei Z. Broder. Some applications of rabin's fingerprinting method. In *Sequences II*, pages 143–152, New York, NY, 1993. Springer New York.

[18] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation*, 79:258–272, 2014. URL https://www.sciencedirect.com/science/article/pii/S0166531614000790.

[19] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: a fast and efficient content-defined chunking approach for data deduplication. In *USENIX Annual Technical Conference*, 2016.

[20] Yuta Nakamura, Raza Ahmad, and Tanu Malik. Content-defined merkle trees for efficient container delivery. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 121–130. IEEE, 2020.

[21] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011. URL https://hal.inria.fr/inria-00555588.

[22] Git - git-notes Documentation. https://git-scm.com/docs/git-notes, . Last Accessed: 2022-08-09.

[23] Lucy Lu Wang, Kyle Lo, Yoganand Chandrasekhar, Russell Reas, Jiangjiang Yang, Doug Burdick, Darrin Eide, Kathryn Funk, Yannis Katsis, Rodney Michael Kinney, Yunyao Li, Ziyang Liu, William Merrill, Paul Mooney, Dewey A. Murdick, Devvret Rishi, Jerry Sheehan, Zhihong Shen, Brandon Stilson, Alex D. Wade, Kuansan Wang, Nancy Xin Ru Wang, Christopher Wilhelm, Boya Xie, Douglas M. Raymond, Daniel S. Weld, Oren Etzioni, and Sebastian Kohlmeier. CORD-19: The COVID-19 open research dataset. In *Proceedings of the 1st Workshop on NLP for COVID-19 at ACL 2020*, Online, July 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.nlpcovid19-acl.1.

[24] Git annex. https://git-annex.branchable.com/, . Last Accessed: 2022-11-15.

[25] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813, RFC Editor, June 1995. URL https://www.rfc-editor.org/info/rfc1813.

[26] Mahima Pushkarna, Andrew Zaldivar, and Oddur Kjartansson. Data cards: Purposeful and transparent dataset documentation for responsible ai. In *2022*

ACM Conference on Fairness, Accountability, and Transparency, FAccT '22, page 1776–1826, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393522. doi: 10.1145/3531146.3533231. URL https://doi.org/10.1145/3531146.3533231.

[27] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J Elmore, Samuel Madden, and Aditya G Parameswaran. Datahub: Collaborative data science & dataset version management at scale. In *CIDR*, 2015.

[28] Ashvin Agrawal, Rony Chatterjee, Carlo Curino, Avrilia Floratou, Neha Gowdal, Matteo Interlandi, Alekh Jindal, Kostantinos Karanasos, Subru Krishnan, Brian Kroth, et al. Cloudy with high chance of dbms: A 10-year prediction for enterprise-grade ml. *CIDR*, 2020.

[29] Duckdb - an in-process sql olap database management system. https://duckdb.org/. Last Accessed: 2022-11-20.

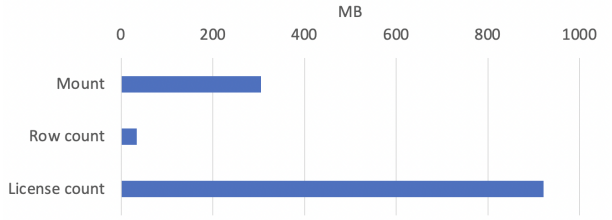
[30] Mark Raasveldt and Hannes Mühleisen. Data management for data science - towards embedded analytics. In *CIDR*, 2020.

[31] Laion-400-million open dataset. https://laion.ai/blog/laion-400-open-dataset/, . Last Accessed: 2022-11-20.

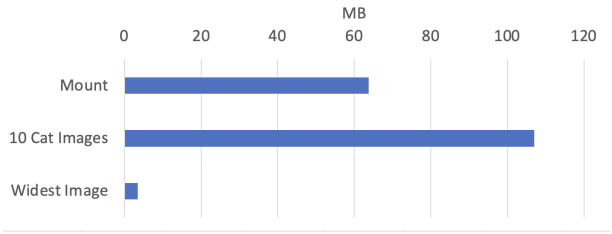
[32] Sqlite home page. https://www.sqlite.org/index.html, . Last Accessed: 2022-11-20.

[33] Exploring 12 million of the 2.3 billion images used to train stable diffusion's image generator. https://waxy.org/2022/08/exploring-12-million-of-the-images-used-to-train-stable-diffusions-image-generator/, . Last Accessed: 2022-11-20.

[34] sqlite3vfshttp: a go sqlite vfs for querying databases over http(s). https://github.com/psanford/sqlite3vfshttp, . Last Accessed: 2022-11-20.



(a) Laion400M Parquet Queries with DuckDB



(b) LaionAesthetic SQLite Queries

Laion400M Parquet Queries with DuckDB

```
select COUNT(*) from 'data/*.parquet'
select LICENSE, count() from 'data/*.parquet'
                        group by LICENSE
```

LaionAesthetic SQLite Indexes

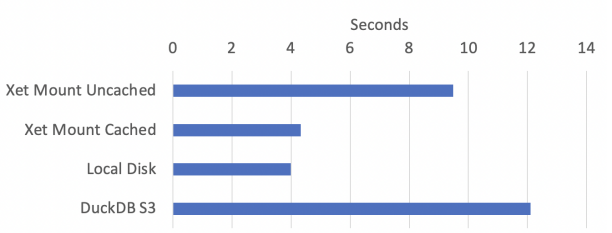
```
create index idx_width on images(width)
create index idx_height on images(height)
create index idx_width_height on images(width,height)
create index idx_url on images(url)
CREATE VIRTUAL TABLE images_text using fts5(url, text)
```

LaionAesthetic SQLite Query

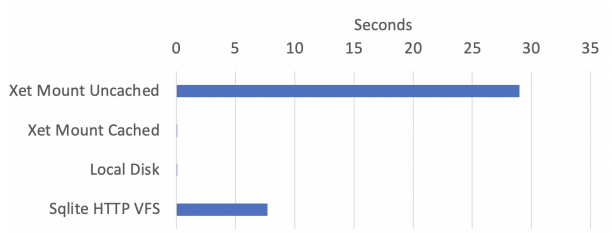
```
select * from images_text
        inner join images
        on images_text.url==images.url
        where images_text.text MATCH 'cat' LIMIT 10

select * from images where
        height == (select MAX(height) from images)
```

(c) Queries and Indices

Figure 6: Total MB downloaded per query for each dataset. `xet mount` was configured with prefetch disabled and 1MB cache block size to optimize for random access. (a) SQL queries using DuckDB [29, 30] performed on a 54GB Xet dataset of Parquet files obtained from [31]. The LAION-400M dataset comprises of Image URLs, text descriptions and other image metadata including a license. As Parquet is columnar, columnar queries are efficient and only a small fraction (2.3%) of the dataset needs to be downloaded to obtain a license distribution. (b) SQL queries on a 9GB SQLite [32] database built from a 12M image subset of the Laion-Aesthetic dataset [33]. Appropriate column indexes are created to avoid a complete table scan for the queries tested. (c) Queries and indices used for for the Parquet and SQLite queries in (a) and (b).



(a) Laion400M Parquet License Count Benchmark



(b) LaionAesthetic SQLite Cat Images Benchmark

Figure 7: All benchmarks performed on a t2.xlarge AWS instance with 4 vCPUs and 16GB RAM. (a) Performance of the Parquet license count query (Fig. 6) comparing query runtime: (i) immediately after mount (uncached), (ii) subsequent runs (cached), (iii) from local disk, (iv) directly from S3 bucket using DuckDB's native connector. Linux page caches were flushed prior to every query. Since the parquet page size is large and DuckDB parallelizes data access, we were able to obtain very good performance for Parquet queries even outperforming DuckDB's native connector by 21%. Once accessed, our cached performance is comparable to direct local disk performance.
(b) Performance of the SQLite 10 Cat Images query (Fig. 6) comparing query runtime: (i) immediately after mount (uncached), (ii) subsequent runs (cached), (iii) from local disk, (iv) directly from S3 bucket using a SQLite VFS HTTP Connector [34]. As the SQLite default page size is small (4K), our 1MB cache block size is far too large resulting in a nearly 4x slowdown for the uncached query compared with the SQLite VFS connector. However, once cached, performance is on-par with local disk (<0.1s).