DAC-MAN: Data Change Management for Scientific Datasets on HPC systems

Devarshi Ghoshal, Lavanya Ramakrishnan, Deborah Agarwal
Lawrence Berkeley National Laboratory, Berkeley, CA
Email: {dghoshal, lramakrishnan, daagarwal}@lbl.gov

***Abstract*—Scientific data is growing rapidly and often change due to instrument configurations, software updates or quality assessments. These changes in datasets can result in significant waste of compute and storage resources on HPC systems as downstream pipelines are reprocessed. Data changes need to be detected, tracked and analyzed for understanding the impact of data change, managing data provenance, and making efficient and effective decisions about reprocessing and use of HPC resources. Existing methods for identifying and capturing change are often manual, domain-specific and error-prone and do not scale to large scientific datasets. In this paper, we describe the design and implementation of DAC-MAN framework, which identifies, captures and manages change in large scientific datasets, and enables plug-in of domain-specific change analysis with minimal user effort. Our evaluations show that it can retrieve file changes from directories containing millions of files and terabytes of data in less than a minute.**

I. INTRODUCTION

Experimental and observational data from science user facilities are growing rapidly in data size and complexity, requiring the use of computational, networking and storage infrastructure at HPC centers. However, existing HPC platforms and tools are designed for high concurrency MPI workloads and significantly simpler data needs. A fundamental challenge with scientific data on HPC systems is that it is frequently updated due to the changes in instrument configuration, software updates, quality assessments or data cleaning algorithms. Data producers [1], [2] often publish data as different data releases, which are annotated with high-level description about the changes. However, they often lack information at a level that is necessary to make decisions on the impact of data change. For example, there is a lack of information on the number of files changed, types of changes, amount of data change. Scientific users often re-run pipelines with entire datasets due to lack of information about the exact data changes.

In order to deal with data changes, users reprocess the complete new dataset, and develop ad-hoc scripts that are often manual, domain-specific and error-prone. These manual solutions do not scale for large scientific datasets. Thus, it is important to capture and understand the changes in data in order to analyze the impact on dependent results and scientific discoveries, and make efficient and effective resource usage.

Today, there are a limited number of tools and libraries used to detect data changes [3], [4]. Unix `diff` is a simple tool to detect file changes. Python provides a few libraries to allow users to detect changes in data and filesystem [5], [6]. However, these tools can only work effectively with text files, and are not able to handle different data and file formats representing different scientific datasets. Version control systems also provide users an infrastructure to manage and track changes [7], [8]. However, they are mostly used to keep track of source-code changes, which also contains textual data. In addition, different file formats and scientific data types add to the complexities of analyzing data change. Existing solutions are not designed to scale for scientific datasets with large and complex files and directory structures. Additionally, the performance of existing tools deteriorates significantly as the number of files increase (Figure 14 in Appendix).

In this paper, we propose an end-to-end methodology to track changes in scientific datasets. We design and develop a framework, called **DAC-MAN**[1], that allows users to efficiently and effectively identify, track and manage data change and associated provenance in scientific datasets. DAC-MAN seamlessly identifies and captures changes between multiple versions of a dataset using efficient indexing and caching mechanisms. The framework detects changes of different types and granularities. DAC-MAN also allows users to plug-in domain-specific data change analysis scripts. The framework's current implementation provides a command-line utility that is used to track and capture changes offline. However, the framework is flexible such that it can also be setup for real-time data comparison. DAC-MAN is a foundational block that is needed to manage data and workflows from data-intensive sciences on supercomputers. It can be integrated with existing data management frameworks [9], [10], providing them the necessary information required to efficiently manage data on large HPC systems. The knowledge of data change provided by DAC-MAN can also be used to selectively process data, enabling workflow managers and resource managers to efficiently and effectively use HPC resources.

The novelty of DAC-MAN is in the end-to-end methodology of tracking and managing data change using indexing and caching on HPC systems. Specifically, we make the following contributions in this paper:

- We describe the design and implementation of DAC-MAN.
- We describe our methodology to compare and capture changes at different levels of a scientific dataset leveraging a bi-directional indexing and caching.

[1] DAC-MAN stands for **DA**ta **C**hange **Man**agement

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00

- We evaluate the performance of DAC-MAN on synthetic and scientific datasets.

The rest of the paper is organized as follows. Section II presents an overview of use cases and design considerations for efficient change capture. Section III presents the design and implementation of DAC-MAN. We present our results in Section IV. We discuss related work in Section V and the conclusions in Section VI.

II. BACKGROUND

In this section, we discuss a sample workflow and key design considerations for a change capture framework.

A. Sample Workflow

We illustrate the challenges with data changes using the NASA MODIS data. MODIS (or Moderate Resolution Imaging Spectroradiometer) is an instrument aboard the Terra and Aqua satellites. The data collected by MODIS helps the study of global dynamics and processes occurring across land, oceans, and lower atmosphere and is used by many different scientists. Users download the data from the website and perform complex pre-processing steps before using the data. Previous work has highlighted the challenges with data procurement and processing due to difficulties in volume, size of data and scale of analyses [11]. For example, a year's data contains tens of thousands of files and is ≈1 TB. High performance file systems are designed to handle large number of small files, resulting in performance issues in the workflow. The pre-processing step to reproject the data to a specific resolution and coordinate system can take ≈5.8K compute hours on a NERSC system [11].

The datasets are often updated as quality assessment and quality control steps are performed. However, there is no description of changes available with new versions of the dataset. Additionally, there is limited or no provenance that captures the source and reasons behind the change. Scientists might use simple tools like diff to inspect the changes but often the process is too tedious or provides incomplete information. Hence, in the absence of any or limited provenance, scientists often choose one of two directions for handling the change – delay the processing or download all the data and reprocess. For example, they may choose to delay the process of updating the downstream products due to the complexity of the processing resulting in mismatch between the versions of the products that is often hard to track due to lack of provenance. If and when they choose to use the new version of the data, they download all the data, reprocess it and regenerate the outputs. This results in a waste of compute resources and heavy use of the file system on supercomputing resources.

B. Design Considerations

In this section, we highlight the design considerations for a scalable data change management framework.

Scalability. One of the primary design considerations for a change management framework is its ability to scale with larger datasets. It also needs to allow users to scale their data comparisons from desktops to supercomputers.

Ability to generate and share data change summaries. Data producers often need to summarize and share changes for data releases to the users. Hence, the framework should be capable of summarizing and sharing the changes to users in a meaningful way.

Classification of different types of change. Users often need to distinguish between metadata and data-level changes in scientific datasets. It is important to classify the changes into different types, allowing users to make informed decisions based on the importance and significance of the change. This allows users to selectively process the data. Hence, the framework should be able to define the different types of changes and classify the data.

Compare remote repositories. In science environments, often two datasets are not co-located on the same system. Different versions of the dataset may exist on two remote locations, e.g., two supercomputing facilities. It is important to capture the changes between such datasets without transferring all the data over the network to a common location.

Support domain-specific change analysis. Scientific datasets come in different file and object formats. Such datasets often need domain-specific analysis for understanding the semantic changes in the data. The framework should allow users to plug-in and scale external scripts for data change comparisons.

Compatible with existing software ecosystems. Software ecosystems in scientific collaborations and supercomputing centers can be diverse and complex. It is critical that the tool is compatible with existing software stacks in collaborations. Data repositories for large collaborations are distributed on different systems or filesystems and under control of different users or groups. A data change framework cannot disrupt these social structures and/or assume it can own and control the data.

III. DESIGN AND IMPLEMENTATION

DAC-MAN provides the end-to-end methodology and workflow needed to effectively and efficiently determine, track and manage change across datasets. DAC-MAN is designed and implemented to work with current filesystems and software ecosystems in scientific collaborations and HPC systems. It uses a parallel indexing technique to determine the changes in the data that is then used for quick look-ups for user queries. Also, since data change queries are likely to be repeated across directories and subdirectories, DAC-MAN manages a cache that stores pre-computed change results. The combined approach of parallel indexing and caching enables us to provide efficient data change comparisons over large datasets that are increasingly using HPC systems.

Figure 1 shows the high-level architecture of the DAC-MAN framework. The framework has two main components – a) *change tracker*, and b) *query manager* – that track the changes in data and manage the data change related queries respectively. In the interactive mode, a user queries for the changes by specifying two directories that contain

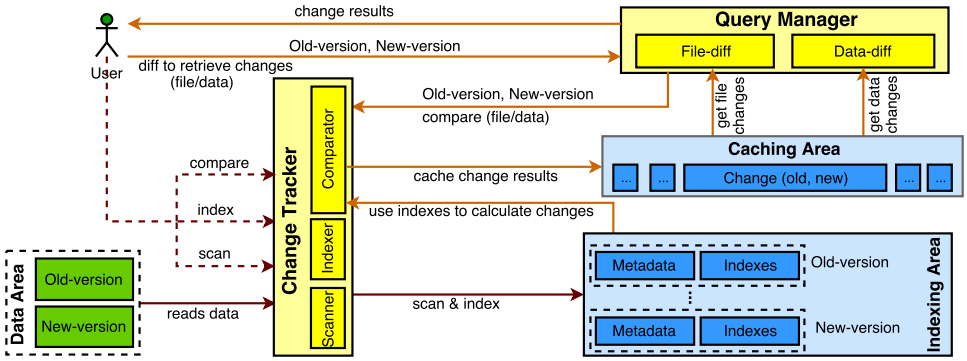


Fig. 1: DAC-MAN architecture. Users query for the changes between two directories that contain different versions of a dataset. The *query manager* requests the *change tracker* to compare the files in the directories. The directories are indexed and the indexes are saved in the *indexing area*. The change tracker uses the indexes to calculate changes in the files and saves the change information in the *caching area*. The query manager returns the changes from the caching area to the user.

the different versions of a dataset. The query manager looks for the requested change in the caching area. The caching area maintains any previously calculated change information between the two directories. If the change information is already in the cache, it returns the results back to the user. Otherwise, the query manager requests the change tracker to compare the files in the specified directories. The change tracker fetches the metadata and indexes for the files if they exist. If the index does not exist, it generates the index for the data. The change tracker uses the indexes to do a comparison between the files, and saves information about the file changes in the caching area. The use of index for comparison ensures that our comparison is efficient and 'fast'. A user can request a) a higher-level summary of file changes (i.e., which files changed) or b) a fine-grained comparison of two files (i.e., what has changed between two files). If the user queries for file changes, the query manager retrieves the changes directly from the cache. If the user queries for data changes, the query manager fetches the changed files, and requests the change tracker to compare the data in files. The change tracker compares the data and returns the results back to the user through the query manager.

The change detection steps in DAC-MAN consist of a collection of sequential and parallel tasks, resulting in a 'change capture workflow' that includes steps for scanning of the directories, parallel indexing of the data, comparison of the data and parallel diff (More details in Appendix C.)

A. Indexing

DAC-MAN creates a *bi-directional* index, which contains both forward and reverse indexes on the dataset. In other words, the forward index maps a file to the hash of its data, and the reverse index maps the data hash to the file. The forward index is used by DAC-MAN to compare files with the same relative paths in two dataset versions. It is used to check if a file has been modified or is unchanged. The reverse index is used to compare the data hashes to find out any path-related changes in the files. The use of bi-directional indexes allow DAC-MAN to capture both data and metadata changes in the dataset. In addition, indexing in DAC-MAN also creates a dictionary object where keys are the file names, and the values are lists of all different paths that contain the files of the same name.

DAC-MAN builds the file indexes in parallel. On a single node, DAC-MAN uses Python multiprocessing and on a multi-node cluster, it uses MPI for parallel indexing. DAC-MAN maintains an internal queue to keep track of the names of files to be indexed. Each process fetches a file name, builds the index, and removes the associated file entry from the queue. The workers continue this process until all the files in the queue are indexed. Finally, DAC-MAN saves these indexes on the disk that is used for efficient change comparison.

DAC-MAN recursively indexes all the directories and subdirectories of a dataset. This alleviates the users from indexing the subdirectories and computing the changes for each subdirectory of a dataset separately. DAC-MAN maintains all the indexes in a separate indexing area on the filesystem. Since the indexes are saved directly on the filesystem, they are extremely light-weight and allows for fast comparison because there are no associated overheads. Since the indexing area in DAC-MAN is a separate directory where the indexes are stored, users can easily copy the indexes between multiple machines. This allows for comparing datasets that are not co-located on the same system.

B. File Comparator

DAC-MAN implements a file comparator that can compare arbitrarily large number of files and directories. When two directories are compared, files in the two versions are classified into the following types – a) added, b) deleted, c) modified, d) metadata-only and e) none/unchanged. DAC-MAN uses a set of rules to classify the files into these different types.

A file is classified as unchanged only if its file properties (path and name) and data remain unchanged. A file is added/deleted, if no corresponding comparison is found in the other version of the dataset, i.e., there is no matching file path, name or data for the file in the other version. A file is classified as modified if its data changes, but one or more file properties (i.e., metadata) remain the same. Finally, a file is said to have metadata-only changes if there is a file with the same data, but with one or more different file properties. Appendix D lists the rules for classifying changes into different types.

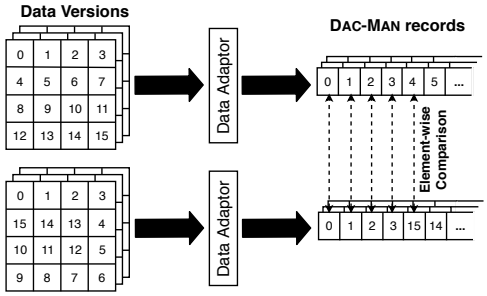


Fig. 2: DAC-MAN reshapes multi-dimensional scientific datasets into one-dimensional arrays, and uses element-wise comparison between the corresponding records.

There are also cases when a direct comparison of a file is not easily available. For example, two dataset versions may contain multiple files that have the same names, but different paths. These files may be organized in separate subdirectories. For example, a dataset may contain a file with name `foo` under three different paths – `/dataset/foo`, `/dataset/some/foo` and `/dataset/other/foo`. Since files with different paths and same names may result in multiple comparisons, DAC-MAN implements a *conflict resolution* strategy to do one comparison per file for performance reasons. It makes use of the dictionary object created during indexing, which contains the mapping of file names to all the associated paths. DAC-MAN uses this dictionary only when a file does not have identical paths, but have the same name in the two versions of the dataset. DAC-MAN marks these files as conflicts. The conflict resolution strategy then uses a heuristic to yield the best possible match using the edit distance [12] between the file paths. It fetches all the associated paths of a file from the dictionary object that is created during indexing of a dataset, and selects the path with the minimum edit distance. In case, more than one path has the minimum edit distance, DAC-MAN selects the first path in the list. Once the corresponding file path is selected for the conflicted file, DAC-MAN classifies the file as metadata-only change and saves the compared file path in the other version.

The heuristic uses the indexes rather than the data in files for faster conflict resolution and file comparison. The heuristic also limits change identification to only one comparison per file. The other alternative is to compare a file to all the associated paths with matching file names. However, this is not a scalable solution as it may lead to mXn comparisons, where there are m and n occurrences of a file with the same name in two versions of a dataset respectively.

The classification of changes allows DAC-MAN to simplify and speedup change retrieval when users query for the changes. DAC-MAN also uses this classification to calculate data changes for the files that are modified. However, users can explicitly specify two arbitrary files to be compared, and DAC-MAN uses the relevant data comparator for calculating the data changes between files.

C. Data Comparator

As discussed earlier, scientific datasets can be represented using different file formats and structures including comma separated values, tables, arrays etc. DAC-MAN implements several "data adaptors" to support data comparison for different scientific datasets. The currently implemented adaptors use specific libraries [13], [14], [15] to read and transform data from several scientific data formats [16], [17] into a common format that is understood by the DAC-MAN data comparator. Additional data adaptors can be similarly implemented for other scientific data formats.

The common format that the data comparator understands is called a "DAC-MAN record". A DAC-MAN record is a flattened representation of any arbitrary multi-dimensional array. Each element of a DAC-MAN record contains a key and one or more values that the key uniquely identifies. This allows for comparing the values by matching corresponding keys in two records. A DAC-MAN record is a flattened one-dimensional array, where the index of an array element is the key and the array element is the value. Thus, the data comparison is simply reduced to an element-wise comparison of the two arrays as shown in Figure 2. Other data comparison algorithms for finding changes in data can also be implemented in the data comparator. However, many of the existing algorithms [4], [18] have polynomial time complexity and can be extremely slow for large scientific datasets.

A domain-specific change analysis often uses statistics [19] or other techniques [20], [21] that might apply to a specific dataset like XML, images, etc. DAC-MAN allows users to develop their own data comparator scripts and use them as plug-ins. The script needs to accept the file names as command-line arguments. When the script is plugged-in, DAC-MAN uses the pair of files to be compared as the command-line argument to the script. The plug-in feature allows users to efficiently scale-up domain-specific scripts. since DAC-MAN parallelizes data change calculation using MPI.

D. Caching

DAC-MAN uses the indexes to compute file and data changes. Once the changes between two datasets are computed, DAC-MAN saves the results in a *cache*. The cache acts like a staging area (similar to Git) in DAC-MAN for faster retrieval of file changes, if users query for the changes multiple times. It only saves the metadata about the file changes, associating the type of change (as classified in Table II) for each file in the dataset. When a user queries for data changes, DAC-MAN first retrieves the list of modified files from the cache, and then compares them using the data comparator. The cache is persistent because the results are saved on disk. Hence, it is only limited by the available space of the underlying filesystem. The results in the cache are only *overwritten* when files in the directories change. They are never deleted except if the user explicitly clears the cache.

DAC-MAN maintains a list of tags to identify and record change results in the cache. Once a user queries for the changes between two datasets, DAC-MAN saves the query and its results in the cache, and creates a tag for it. Each tag is a unique identifier that corresponds to the hash of the directory paths for which the change is computed in the query.

DAC-MAN creates the tags using the absolute paths of the directories, and hence, creates a unique tag for every dataset irrespective of how the user queries for the changes. If the same query is issued multiple times, DAC-MAN uses the tag to retrieve the change results directly from the cache.

DAC-MAN also uses this cache to directly fetch the changes in the subdirectories of a dataset. DAC-MAN indexes and computes the changes recursively for all the directories and subdirectories of a dataset. Thus, it does not recompute the changes for the subdirectories separately if the top-level directory changes are already cached since subdirectory level changes are a subset of the total dataset changes. When a user queries for the changes of two respective subdirectories in the datasets, DAC-MAN derives the subset of changes by filtering only the results that match the subdirectory paths specified in the query. However, once the subdirectory changes are filtered and derived from the dataset cache, the change results are saved in the cache, and a tag is created that associates the changes to the subdirectories.

The caching area in DAC-MAN is similar to a staging area, where all the changes are saved for fast retrieval, without the need for computing the changes every time a user queries for it. DAC-MAN currently saves all change results as files on the disk. This provides a light-weight solution to capture and analyze changes. However, users can also use relational databases or object stores for complex change analysis and sharing change results. The architecture of DAC-MAN allows users to plug-in components for saving change results into databases and other forms of persistent stores.

E. User Interface

DAC-MAN provides a command-line utility and a programming interface. It is currently implemented in Python and uses Python's *scandir* library as it avoids making unnecessary calls to retrieve file system attributes. We use the *editdistance* library for calculating the minimum edit distance between two strings, which we use for comparing file paths. When the commands are run on a single node, it uses Python multiprocessing for parallel indexing and data change analysis. It allows scaling on multiple nodes through MPI.

Users use the `dacman diff` command to query changes between two datasets. This command allows users to retrieve both filesystem and data changes (using the `--datachange` option). Additionally, users can plug-in their own scripts through this command for doing domain-specific data change analysis. The `diff` command also allows users to scale change analysis across multiple nodes using MPI.

If the changes are not available in cache, then `dacman diff` implicitly indexes the data, computes, and caches the changes. Alternatively, users can also explicitly index the data using `dacman index`. Explicit indexing provides users the flexibility to index the data remotely and compare the changes locally. This is possible since indexes can be copied and used for identifying changes on another machine, without the need for moving the actual data. This is a powerful feature of DAC-MAN as it allows directories and files on separate machines to be compared without necessarily moving all the data to a single place. The `index` command also allows users to explicitly scale out indexing on an HPC system and precompute the indexes to enable fast change detection.

F. Plug-ins

The default data comparator in DAC-MAN may be insufficient for detailed change analysis of complex scientific datasets of different formats. Also, there may be a need for more detailed change analysis based on the results provided by DAC-MAN. Thus, users can define their own data comparators and plug-in scripts in DAC-MAN to compute data changes.

Users can use the command-line utility to specify external scripts as plug-ins for doing data comparisons. Additionally, users can also use the Python programming interface to define and use data comparators for specific datasets.

G. Data Provenance

Provenance information about the datasets is embedded in the different types of changes captured by DAC-MAN. DAC-MAN keeps track of the datapaths and associated metadata information (e.g., ownership, creation date etc.), provenance information is automatically included in the data captured by DAC-MAN. This information can be converted into W3C PROV standards [22] using existing tools like ProvToolBox or Komadu [23]. Additionally for scientific workflows, the change information corresponding to a dataset, as captured by DAC-MAN, can be used to derive the relationship between inputs and outputs of a workflow. For example, if part of the input and output data changes simultaneously for a workflow, DAC-MAN retrieves the corresponding changes, where the changed inputs can be associated with the changed outputs allowing users to track provenance across dataset revisions. Finally, any changes in binary executables or workflow descriptions can also be identified by DAC-MAN, associating the outputs to the corresponding entities that generated them.

DAC-MAN also enables using the provenance information to analyze the impact of changes. Previous work has focused on capturing provenance from execution traces of scientific workflows [24], log files [25] and program instrumentations [26]. These provenance traces can be compared using DAC-MAN for any changes in the workflow parameters, and the associated inputs and outputs. Users can then use DAC-MAN's data comparators, or use domain-specific comparison tools to analyze the changes in these traces for evaluating the impact of data change.

H. Usage on Supercomputers

The command-line utility in DAC-MAN allows users to identify and capture changes 'on demand' from different versions of a dataset. Additionally, it can also be used as a monitoring tool using cron, capturing change information from frequently changing datasets. The programming interface allows users to integrate change identification and capture into their existing data processing pipelines.

Our design process for DAC-MAN is based on user research methods. User research methods enabled us to understand the

Metric	Description
Directory Scan	Time to recursively scan a directory and save file metadata (seconds).
Index Creation	Time to index the data in a directory (seconds).
DAC-MAN diff	Time to retrieve changes between two datasets (seconds).
Total Time	Directory Scan + Index Creation + DAC-MAN diff (seconds)
Index Size	Size of DAC-MAN indexes (MB).

TABLE I: Metrics for evaluating DAC-MAN.

design requirements for DAC-MAN. Subsequently, we have used usability studies to evaluate early prototypes. The usability studies showed that the functionality of the prototypes was well-aligned with the varying needs of the users. Users can use DAC-MAN on their desktops by directly using the command-line interface. Additionally, users can submit DAC-MAN in a batch job script, allowing it to scale to multiple nodes using MPI. Thus, users are able to scale their data change analyses on large scientific datasets. DAC-MAN has also been designed to allow users to easily write their own data comparators or plug-in scripts for specific domains.

IV. EVALUATION

In this section, we evaluate the performance of DAC-MAN in the context of synthetic and scientific datasets. We compare the performance of DAC-MAN with state-of-the-art solutions and evaluate the scalability with increasing data sizes. We also evaluate the performance on different types of data changes. Finally, we evaluate the overheads of using DAC-MAN.

A. Evaluation Setup

We evaluate our system on NERSC's Cori supercomputer. It is a Cray XC40 supercomputer with 1630 compute nodes. Each node has 32 cores and has 128 GB DDR4 2133 MHz memory and four 16 GB DIMMs per socket. Each core has its own L1 and L2 caches, with 64 KB and 256 KB, respectively. We use Cori's Lustre filesystem for storing data, indexes and metadata information. The filesystem has a peak performance of approximately 700 GBps. We evaluate the performance both on the login and compute nodes (at scale). All experiments are repeated three to five times, and we use the mean across the runs for our results. Variability across runs was minimal and hence, not represented in the graphs.

B. Datasets

We use scientific datasets from Sloan Digital Sky Survey (SDSS) and Fluxnet, containing different versions of cosmology and environmental sciences data respectively (more details in Appendix A). Also, we use synthetic datasets to evaluate the effect of different characteristics of data on DAC-MAN. Our evaluations focus on the scalability of DAC-MAN for large scientific datasets. These datasets are selected based on their diversity of types, characteristics and quantity. SDSS datasets contain large number of files, mostly binary and image data with different types and amounts of change in the dataset. Fluxnet contains smaller number of files, commonly in CSV format. Synthetic datasets are randomly generated binary data

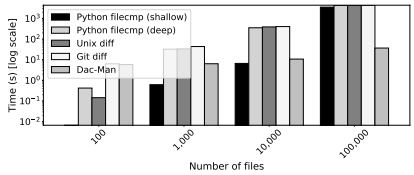
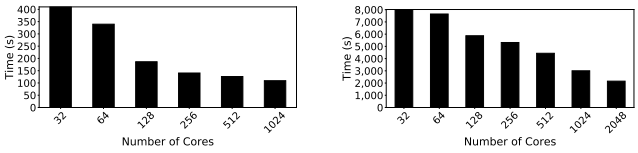


Fig. 3: Comparing DAC-MAN to other state-of-the-art solutions on the synthetic dataset with increasing number of files. DAC-MAN outperforms existing solutions as the amount of data increases.



(a) Indexing Synthetic Dataset (b) Indexing SDSS Redux Dataset

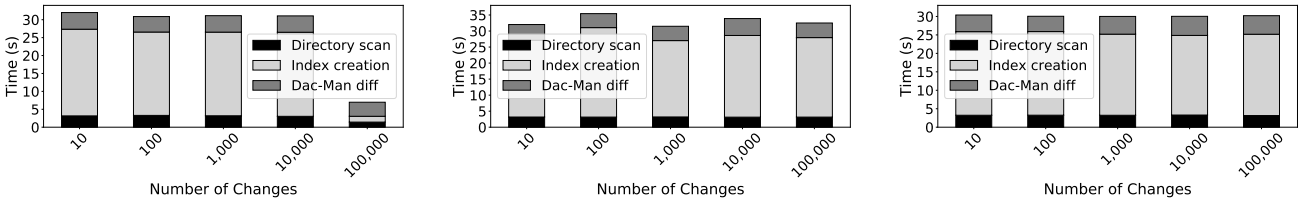
Fig. 4: Scalability of indexing 1 million files of 4 KB each with DAC-MAN. As more compute resources are allocated to DAC-MAN, the indexing time reduces.

in files and the changes are in fixed increments to help evaluate the performance trends in DAC-MAN. Appendix E summarizes the different characteristics of the datasets used in evaluation.

Sloan Digital Sky Survey (SDSS). We evaluate DAC-MAN using three subsets of the SDSS dataset, each with two different versions of the dataset. SDSS datasets primarily consist of FITS files [16], which contain both images and binary data tables. The three SDSS datasets have different characteristics in terms of size of the data and types of changes. The SDSS datasets are – i) Redux, which contains versions 5.6.5 and 5.7.0 from SDSS data release 11 and 12 with a total of ≈9.7 million files (≈4.4 million in one version, and ≈5.3 million in another version) and ≈10 TB total file size between the two versions. ii) Resolve, that contains ≈4.5 million files with a total file size of ≈550 GB divided into two versions corresponding to the SDSS data release 13 generated using the resolve algorithms, and iii) Sweeps, that contains two versions of the reduced sweep imaging catalog data files in SDSS data release 13 with a total of ≈45,000 files and ≈1.2 TB of data. The Redux dataset has a large number of modified files, with very few files that remain unchanged. The Resolve dataset has very few changes, and most of the data remained unchanged. The Sweeps dataset mostly contains metadata-only changes.

Fluxnet. Fluxnet data releases consist of datasets measuring the CO_2, water, and energy fluxes in North, Central and South America. For the Fluxnet dataset, we use the Fluxnet2015 and La Thuile data releases [1]. There are ≈2500 files in the two datasets with a total of ≈1.7 GB of data in size. The Fluxnet data corresponds to the dataset, which requires domain-specific analysis for calculating data changes as all the old files from the La Thuile data release (V_0) have been removed, and new files have been added in the Fluxnet2015 data release (V_1). Specifically, the measurements in Fluxnet2015 are collected by grouping together several years data, whereas for La Thuile data, every year's data is in a separate file. Additionally, the names of the files are changed.

Synthetic Dataset. The synthetic datasets are generated by creating files using random data. Each synthetic dataset has



(a) Number of files deleted (b) Number of files modified (c) Number of files with metadata-only changes

Fig. 5: Time to complete each stage in DAC-MAN for a synthetic dataset of 100,000 files with different types of changes. The total time to identify and retrieve changes remain unchanged, except when all the files are deleted in one version of the dataset.

two versions to compare. The total number and size of each synthetic dataset is dependent on the goal or dimension of the specific experiment. The different dimensions based on which the experiments are executed are – i) type of change (Appendix D), ii) number of files changed, iii) total number of files in a dataset, iv) individual file size, and v) the directory structure and layout of files across different subdirectories. For a systematic evaluation of DAC-MAN, each synthetic dataset is generated by varying only one dimension and keeping the others fixed. We use the synthetic dataset to understand DAC-MAN's performance with different dataset characteristics to provide a systematic and controlled evaluation of DAC-MAN under several changing parameters.

C. Tools and Metrics Used for Comparison

In this section, we list the tools and metrics used for evaluating DAC-MAN. We compare DAC-MAN with Unix diff utility and Git [27]. Git allows users to identify changes in two directories through its *diff* option. Users can specify the `--no-index` flag to compare any two arbitrary directories. Python is being increasingly used in HPC environments and thus, we also compare the performance of DAC-MAN with Python's filecmp library. Specifically, we evaluate the performance using 'shallow' and 'deep' options in filecmp. The 'shallow' option only uses the file metadata to identify files that are changed. The 'deep' option compares the contents of the files to determine the changes. We also compare the performance of identifying data changes using the DAC-MAN's data comparator and Python's difflib library.

We evaluate the performance of DAC-MAN using a specific set of metrics listed in Table I. We use the total runtime of DAC-MAN that includes the time to scan the data, index them, and save and retrieve the change results from the cache. We measure the time to retrieve changes with different frequencies in DAC-MAN to understand the effect of caching. Finally, we measure the storage overheads of creating indexes.

D. Comparison to Existing Tools

Figure 3 compares the performance of DAC-MAN to existing tools for identifying file changes with increasing number of files. We compare the performance on synthetic dataset, where each file is 4KB in size, and the tests are run on a single node (=32 cores). The Y-axis is in log scale. For DAC-MAN, we use the total time to compute and retrieve changes including directory scan and indexing.

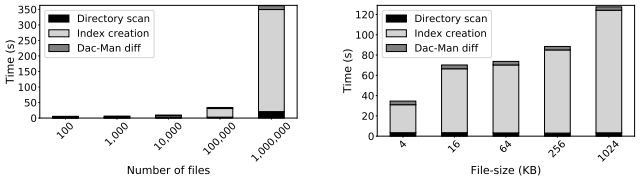
DAC-MAN performs significantly faster than other state-of-the-art tools. For 100,000 files, DAC-MAN is $\approx 120\times$ faster than Git and Unix diff, and filecmp's deep comparison, and $\approx 100\times$ faster than filecmp's shallow comparison. This is because DAC-MAN takes advantage of the multicore architecture, creating the indexes in parallel, and then using the indexes to compare files. This is significantly faster than individually comparing the files, which involves opening the files, reading them and comparing the data. The performance of the shallow comparison deteriorates with increasing number of files because it uses large number of metadata operations on the Lustre filesystem. This creates a bottleneck because of the performance limitation of Lustre's metadata server. In contrast, DAC-MAN compares files based on indexes rather than data and does a fast scan using Python's scandir library which avoids making unnecessary calls to the metadata server.

For smaller datasets, the difference between DAC-MAN and the other tools is not significant because of the indexing overheads. The total time to identify the changes for smaller datasets is generally small. Hence, the performance gains of using the indexes for comparison, is overshadowed by the overheads of creating the indexes in DAC-MAN. For larger datasets, using indexes in DAC-MAN for comparing the datasets improves the performance by three orders of magnitude ($\approx 3000\times$, only for the 'diff' step). Hence, for larger datasets DAC-MAN performs better than other tools even with the indexing overheads.

E. Scalability of DAC-MAN Indexing

Figure 4a shows the time taken to index the synthetic dataset with 1,000,000 files each of 4 KB in size with increasing number of compute resources. As can be seen from the figure, the time taken to index the files reduces as we allocate more cores for indexing. On a single node (=32 cores), DAC-MAN uses Python's multiprocessing module to parallelize index creation. On a multi-node HPC cluster ($\geq$64 cores), DAC-MAN uses MPI to parallelize indexing. There is a $3\times$ speedup when we allocate 1024 cores for indexing 1,000,000 files using DAC-MAN.

Figure 4b shows similar scalability for SDSS Redux dataset as that of synthetic dataset. The indexing time starts to decrease as more resources are allocated to DAC-MAN. There is a $4\times$ speedup when 2048 cores are allocated to do parallel indexing. The time to index the datasets includes reading the files in parallel and computing their data hashes. The results show that the use of a parallel file system (like Lustre)



(a) Changes in number of files (b) Changes in file sizes

Fig. 6: Time to complete each stage in DAC-MAN for the synthetic datasets with varying number of files (each of size 4 KB) and file sizes (total of 200,000 files). The total time increases as the number of files and file sizes increase due to indexing overheads.

enables parallel I/O that allows for indexing performance to be improved as more resources are allocated to DAC-MAN. However, the results show that performance does not scale at the same rate, when more compute resources are allocated since DAC-MAN indexing is I/O intensive. Indexing involves reading several files in parallel gets affected by the I/O limitations of the underlying filesystem. Previous studies have shown that reading large number of files in parallel affects the metadata server of the parallel file system (Lustre) [28]. Additionally, the performance is also limited by the I/O bandwidth of the underlying filesystem [29]. Some of these performance bottlenecks can be eliminated by using high-speed storage systems, like Burst Buffers where the metadata server overloading can be minimized [30], [31].

F. Effect of Number of Changes on Total Time

Figure 5 shows the total time to identify and retrieve changes when there are different types of changes in the dataset. X-axis represents the number of changed files in the revision directory with respect to the base directory. For this experiment, we use the synthetic dataset with 100,000 files in the base version, where each file is 4 KB in size.

As can be seen from Figure 5a, the total time to identify and retrieve changes is extremely small when all the files in revision are deleted. This is because the total number of files to be indexed in the revision directory is reduced to zero. For the other cases in Figure 5a, the number of deleted files is comparatively small ($< 10\%$) as compared to the total number of files in the dataset. Hence, the change detection time is unaffected by the deletions.

Figure 5b shows that number of file modifications does not change the time to identify and retrieve change information if the number of files and file size remains the same. Similarly, Figure 5c shows that the amount of metadata-only changes (where the data remains unchanged, and the file properties change) does not impact the time to identify and retrieve change information. Hence, data or metadata changes by itself, have no effect on the total runtime of DAC-MAN.

G. Effect of Indexing Overheads on Total Runtime

In this section, we evaluate the indexing overheads of DAC-MAN with different file characteristics. Figure 6a shows the total time to identify and retrieve changes when the number of files change in the synthetic dataset. Each file is 4 KB in size. The majority of the total time is used in indexing the datasets since these experiments are run on a single node with 32 cores. It is also important to note that the time to scan the directories and retrieve the data changes increase with increasing number of files. However, directory scan time is very small ($\approx$ 12 seconds) as compared to the indexing time ($\approx$ 400 seconds) for 1,000,000 files.

Figure 6b shows the time to identify and retrieve changes when the file size changes in the synthetic dataset. X-axis represents the individual file size. As the file size increases, the change detection time increases due to increase in indexing time. The indexing time increases because DAC-MAN indexes the files based on the hash of the data. This requires reading all the data in the file and hence, with increasing file size more data is to be read and hashed, to build the index.

Figure 7 shows the time to identify and retrieve changes in SDSS and Fluxnet datasets. Since, the Redux dataset in SDSS has a large number of files (approx. 5 million files in each version, around 10 TB of data), directory scanning and indexing take most of the change detection time. However, once the data has been indexed, the time to retrieve changes is significantly small ($\approx$ 32 seconds). For smaller datasets (e.g.,Fluxnet and Sweeps), the indexing and scanning times are significantly lower than that of the Redux dataset because there are fewer files to be indexed.

H. Effect of Caching on Retrieving Changes

Figure 8a shows the time to retrieve the changes when the number of files in the synthetic dataset changes. Retrieving the changes for the first time increases with the increasing number of files. This is because DAC-MAN computes the changes and saves it in the persistent cache when a diff is requested for the first time. However, if a user queries for the changes in datasets which are already computed, DAC-MAN retrieves it in constant time from the cache.

Figure 8b shows that the change retrieval time in DAC-MAN is independent of the size of data. DAC-MAN uses the indexes to compute and retrieve the changes and does not use the data in files directly to compute the changes.

Figure 9 shows the query response time for retrieving changes at different subdirectory levels. The results show the time to retrieve changes from subsequent subdirectories of a synthetic dataset. For this experiment, we use the synthetic dataset containing a total of 1,000,000 files. There are four subdirectories with 200,000 files in each subdirectory. The X-

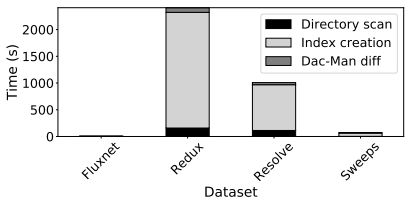
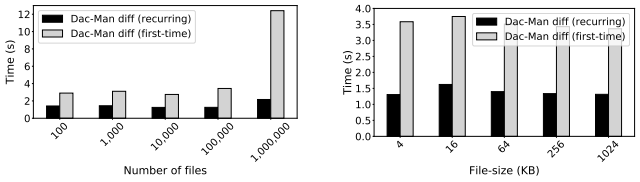


Fig. 7: Time to complete each stage of change detection in DAC-MAN with SDSS (Redux, Resolve and Sweeps) and Fluxnet datasets.



(a) Changes in number of files (b) Changes in file sizes

Fig. 8: Time to retrieve changes from the synthetic datasets with increasing number of files (each of 4 KB in size), and increasing file sizes (for 100,000 files). Time to retrieve changes for recurring queries is constant and significantly less due to caching.

axis represents the level of the subdirectory, where level 0 refers to the top directory, level 1 refers to a subdirectory within the top directory and so on. Change results can be directly retrieved for any level of subdirectory using the top-level directory indexes since indexing the top level directory with DAC-MAN indexes all the files and subdirectories recursively. Figure 9 shows that as we query for changes from deeper subdirectories for the first time, the retrieval time reduces. This is because the total number of files decreases with the increasing depth of subdirectories. However, as is the case with other change retrieval queries, if the subdirectory changes are already computed and cached, then the recurring identical query yields the results in constant time.

Figure 10 shows the time to retrieve changes from the scientific datasets. For large datasets like Redux, the change results are retrieved in $\approx$ 30 seconds when DAC-MAN precomputes the changes and the results are saved in the cache. When the changes are computed for the first time, they are retrieved in $\approx$ 85 seconds. For the Resolve dataset, changes are retrieved $10\times$ faster when the results are saved in the cache as compared to retrieving the changes for the first time. For smaller datasets like Sweeps and Fluxnet, the cache does not provide the same level of improvement, but is still at least $2\times$ faster than when the results are retrieved for the first time.

I. Space Overheads

Indexing. Figure 11a shows the amount of space taken by DAC-MAN indexes when the number of total files change in the synthetic dataset. As can be seen from the figure, with increasing number of files, the total index size increases. However, when compared to the total data size, the index size is minimal. For 2,000,000 files and a total 7.6 GB of data,

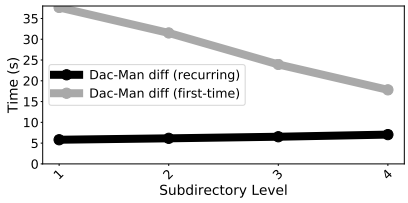


Fig. 9: Time to retrieve changes for synthetic datasets containing 1,000,000 files of 4 KB size each, divided into separate nested subdirectories. Time to retrieve changes first-time decrease as we go deeper into the subdirectories because the number of files per subdirectory reduces.

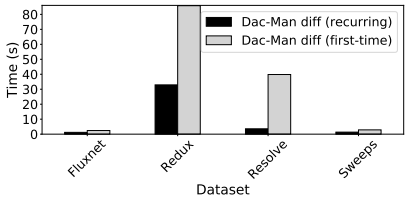
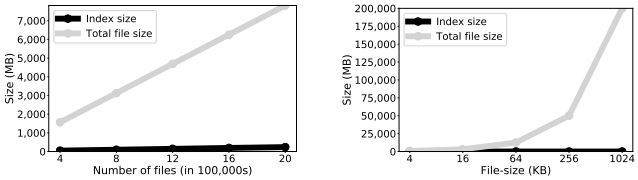
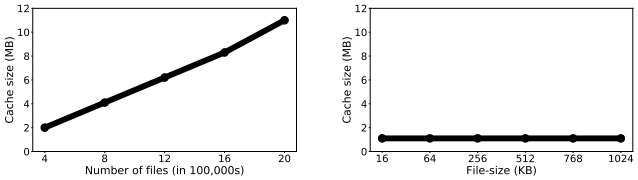


Fig. 10: Time to retrieve changes from SDSS and Fluxnet datasets.



(a) Number of files (b) File-size

Fig. 11: Size of DAC-MAN indexes for increasing size of the synthetic dataset. DAC-MAN indexes have extremely low storage space overheads, and are unaffected by the file size.



(a) Number of files (b) File-size

Fig. 12: Size of the cache for increasing size of the synthetic dataset. DAC-MAN uses the cache to save change results, which is only dependent on the number of files, and not the file size.

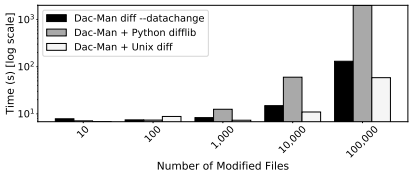


Fig. 13: Using different plug-ins to identify changes in the synthetic dataset with DAC-MAN. DAC-MAN can plug-in and scale any user script to identify and capture data changes from large scientific datasets.

the total index size is 236 MB. The index sizes stay small since they only save the hash of the data in a file, which is substantially small compared to the actual data in file. Also, as the data hashes are fixed in size, changes in file size do not affect the size of DAC-MAN indexes as shown in Figure 11b.

Caching. Figure 12 shows similar trends for caching in DAC-MAN as that of indexing, i.e., cache size increases as the number of files in the datasets increase. However, if the number of files remain the same, but their sizes change, the cache size remains the same. DAC-MAN only saves filenames and associated change types in the cache, which is only dependent on the total number of files in the datasets.

J. Computing Data Changes

Figure 13 compares the results of plugging in different diff tools with DAC-MAN for identifying data changes. The figure shows the results of detecting data changes for varying degree

of changes in a synthetic dataset with 100,000 files of 4 KB size each on a 32 core node.

The results also show that Unix diff performs faster than DAC-MAN's in-built data comparator and Python's difflib. The current implementation of the DAC-MAN data comparator simply compares two one-dimensional arrays with O(n) time complexity, n being the size of the smaller array. However, there are overheads of transforming the data from any arbitrary file into 'DAC-MAN records'. This extra processing for obtaining richer change results affects the overall performance while calculating data changes in large files. For non-text based files, Unix diff simply shows if the file is different or not. In contrast, DAC-MAN can identify the changed values in the datasets using DAC-MAN records.

V. Related Work

Change Identification. Unix diff [32] is the most commonly used tool for finding changes in text files. Version control systems like Git [7] and Subversion [8] use the text diff algorithm to find changes in small text files efficiently. Git LFS (Large File Storage) [33] provides an efficient solution to manage changes in large files by locally storing the contents in a cache, and using text pointers to the files. However, Git LFS does not provide any means to computing or displaying changes in large files. bsdiff [34] efficiently calculates diffs from executable files. Although this is useful in applying patches to large binary executables, the resulting diffs do not yield meaningful results for analyzing the impact of changes as required in scientific datasets.

Bitemporal databases [35], [36] and slowly changing dimensions [37], [38] have been proposed to keep track of historical changes in data warehouses. Several algorithms have been proposed to capture changes from different types of data like XML [20], hierarchical data [18] and RDF repositories [21]. Machine learning and image processing techniques have been proposed to detect changes in image data [39], [40]. Several other tools exist for identifying changes in files and other types of data [32], [41], [42], [43]. All these algorithms and tools are coupled to specific file and data formats, and are not used for big data. DAC-MAN provides a generic way to capture changes to data and allows for these tools and algorithms for domain-specific change analysis at scale.

Change metrics. Levenshtein distance [12] is a common metric that calculates the minimum number of edits required for changing one string into another. Other edit distances are also proposed to measure the number of changes [44], [45]. Hutchinson metric [46] identifies the differences between two images in fractal image processing. DAC-MAN calculates the number of DAC-MAN records added, deleted and modified. Users can also plug-in domain-specific change analysis scripts into DAC-MAN and define their own metrics.

Data provenance. Data provenance [47] captures the derivation history of data. Provenance keeps track of the entire data generation pipeline to understand the reason for change, and is useful in reasoning about the change, rather than detecting the change itself. Past research has focused on using provenance for identifying changes in storage systems [48], e-science workflows [49], databases [50], and scientific data [51]. Current provenance capture methods include software packages and services [52], [53], provenance-aware solutions [54], [48], and language extensions [55]. Workflow tools [56], [57] integrate provenance capture into the workflow lifecycle in order to analyze changes between multiple runs and the lineage of outputs. DAC-MAN allows users to identify, capture and track data changes that is an integral part of provenance. Missier et al. [24] compare provenance traces to check for workflow reproducibility. Such tools can be integrated with DAC-MAN to explore provenance of the changed datasets.

VI. Conclusions and Future Work

In this paper, we describe the design and implementation of a scalable change tracking and management framework DAC-MAN for large scientific datasets. DAC-MAN uses indexing and caching to minimize the overheads of large-scale data comparison. Our evaluation shows that DAC-MAN can retrieve changes from large datasets in the order of a few seconds. The DAC-MAN framework allows users to compare datasets on separate systems, and lets users specify custom scripts that are automatically scaled for domain-specific change detection and analyses. Our future research will focus on providing abstractions for defining data comparators in DAC-MAN, which will simplify domain-specific change analysis for a large number of scientific data formats.

DAC-MAN provides a strong foundation for detecting and tracking data change that is needed for understanding the effects of changes on downstream data products. DAC-MAN can be used to make data and resource management decisions on HPC systems by integrating with existing data and workflow management systems to provide the necessary information for selectively and efficiently processing data on supercomputing systems. Our user research methods have helped us identify the need for an interactive visual data change exploration interface. The user research methods lead to an iterative R&D process and DAC-MAN will continue to evolve through continuous user feedback and usability studies.

Acknowledgments

This work and the resources at NERSC are supported by the U.S. Department of Energy, Office of Science and Office of Advanced Scientific Computing Research (ASCR) under Contract No. DE-AC02-05CH11231. The authors would like to thank Alex Romosan, Stephen Bailey, Boris Faybishenko, Craig Tull and Juliane Mueller for datasets, suggestions and feedback on the work.

References

[1] "Ameriflux," http://ameriflux.lbl.gov/, 2017.
[2] "SDSS," http://www.sdss.org/, 2017.
[3] R. Love, "Kernel korner: Intro to inotify," *Linux J.*, vol. 2005, no. 139, pp. 8–, Nov. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1103050.1103058
[4] E. W. Myers, "An o (nd) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1, pp. 251–266, 1986.
[5] "Python filecmp," https://docs.python.org/3.6/library/filecmp.html, 2017.

[6] "Python difflib," https://docs.python.org/3.6/library/difflib.html, 2017.

[7] "Github," https://github.com/, 2017.

[8] "Apache subversion," https://subversion.apache.org/, 2017.

[9] R. Schuler, C. Kesselman, and K. Czajkowski, "Data centric discovery with a data-oriented architecture," in *Proceedings of the 1st Workshop on The Science of Cyberinfrastructure: Research, Experience, Applications and Models*, ser. SCREAM '15. New York, NY, USA: ACM, 2015, pp. 37–44.

[10] R. Chard, K. Chard, J. Alt, D. Y. Parkinson, S. Tuecke, and I. Foster, "Ripple: Home automation for research data management," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, June 2017, pp. 389–394.

[11] V. Hendrix, L. Ramakrishnan, Y. Ryu, C. van Ingen, K. R. Jackson, and D. Agarwal, "CAMP: Community Access MODIS Pipeline," *Future Generation Computer Systems*, vol. 36, 2014.

[12] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[13] T. P. Robitaille, E. J. Tollerud, P. Greenfield, M. Droettboom, E. Bray, T. Aldcroft, M. Davis, A. Ginsburg, A. M. Price-Whelan, W. E. Kerzendorf *et al.*, "Astropy: A community python package for astronomy," *Astronomy & Astrophysics*, vol. 558, p. A33, 2013.

[14] E. B. Knudsen, H. O. Sørensen, J. P. Wright, G. Goret, and J. Kieffer, "Fabio: easy access to two-dimensional x-ray detector images in python," *Journal of Applied Crystallography*, vol. 46, no. 2, pp. 537–539, 2013.

[15] A. Collette, *Python and HDF5: Unlocking Scientific Data*. " O'Reilly Media, Inc.", 2013.

[16] J. Ponz, R. Thompson, and J. Munoz, "The fits image extension," *Astronomy and Astrophysics Supplement Series*, vol. 105, 1994.

[17] B. Kemp and J. Olivan, "European data format plus(edf+), an edf alike standard format for the exchange of physiological data," *Clinical Neurophysiology*, vol. 114, no. 9, pp. 1755–1761, 2003.

[18] S. S. Chawathe and H. Garcia-Molina, "Meaningful change detection in structured data," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '97. New York, NY, USA: ACM, 1997, pp. 26–37.

[19] J. Inglada and G. Mercier, "A new statistical similarity measure for change detection in multitemporal sar images and its extension to multiscale change analysis," *IEEE transactions on geoscience and remote sensing*, vol. 45, no. 5, pp. 1432–1445, 2007.

[20] Y. Wang, D. J. DeWitt, and J.-Y. Cai, "X-diff: An effective change detection algorithm for xml documents," in *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, 2003, pp. 519–530.

[21] D. Ognyanov and A. Kiryakov, "Tracking changes in rdf (s) repositories," *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, pp. 373–378, 2002.

[22] P. Missier, K. Belhajjame, and J. Cheney, "The w3c prov family of specifications for modelling provenance metadata," in *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 2013, pp. 773–776.

[23] I. Suriarachchi, Q. Zhou, and B. Plale, "Komadu: A capture and visualization system for scientific data provenance," *Journal of Open Research Software*, vol. 3, no. 1, 2015.

[24] P. Missier, S. Woodman, H. Hiden, and P. Watson, "Provenance and data differencing for workflow reproducibility analysis," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 4, pp. 995–1015, 2016.

[25] D. Ghoshal and B. Plale, "Provenance from log files: a bigdata problem," in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. ACM, 2013, pp. 290–297.

[26] J. Frew, D. Metzger, and P. Slaughter, "Automatic capture and reconstruction of computational provenance," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 485–496, 2008.

[27] L. Torvalds and J. Hamano, "Git: Fast version control system," *URL http://git-scm. com*, 2010.

[28] H. Shan and J. Shalf, "Using ior to analyze the i/o performance for hpc platforms," 2007.

[29] W. Yu, J. Vetter, R. S. Canon, and S. Jiang, "Exploiting lustre file joining for effective collective io," in *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*. IEEE, 2007, pp. 267–274.

[30] A. Kougkas, M. Dorier, R. Latham, R. Ross, and X.-H. Sun, "Leveraging burst buffer coordination to prevent i/o interference," in *e-Science (e-Science), 2016 IEEE 12th International Conference on*. IEEE, 2016, pp. 371–380.

[31] C. Daley, D. Ghoshal, G. Lockwood, S. Dosanjh, L. Ramakrishnan, and N. Wright, "Performance characterization of scientific workflows for the optimal use of burst buffers," *Future Generation Computer Systems*, 2017.

[32] J. W. Hunt and M. MacIlroy, *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.

[33] "Git lfs," https://git-lfs.github.com, 2017.

[34] C. Percival, "Naive differences of executable code," *Draft Paper, http://www.daemonology.net/bsdiff*, 2003.

[35] A. Kumar, V. J. Tsotras, and C. Faloutsos, "Designing access methods for bitemporal databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 10, no. 1, pp. 1–20, 1998.

[36] J. F. Roddick, "A survey of schema versioning issues for database systems," *Information and Software Technology*, vol. 37, no. 7, pp. 383–393, 1995.

[37] R. Bliujute, S. Saltenis, G. Slivinskas, and G. Jensen, "Systematic change management in dimensional data warehousing," Time Center Technical Report TR-23, Tech. Rep., 1998.

[38] R. Kimball and M. Ross, *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[39] T. Celik, "Unsupervised change detection in satellite images using principal component analysis and k-means clustering," *IEEE Geoscience and Remote Sensing Letters*, vol. 6, no. 4, pp. 772–776, 2009.

[40] R. J. Radke, S. Andra, O. Al-Kofahi, and B. Roysam, "Image change detection algorithms: a systematic survey," *IEEE Transactions on Image Processing*, vol. 14, no. 3, pp. 294–307, March 2005.

[41] G. Canfora, L. Cerulo, and M. Di Penta, "Ldiff: An enhanced line differencing tool," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 595–598.

[42] D. Jackson, D. A. Ladd *et al.*, "Semantic diff: A tool for summarizing the effects of modifications." in *ICSM*, vol. 94, 1994, pp. 243–252.

[43] R. S. Gonçalves, B. Parsia, and U. Sattler, "Ecco: A hybrid diff tool for owl 2 ontologies." in *OWLED*, vol. 849, 2012.

[44] R. W. Hamming, "Error detecting and error correcting codes," *Bell Labs Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[45] W. E. Winkler, "String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage." 1990.

[46] V. Drakopoulos and N. P. Nikolaou, "Efficient computation of the hutchinson metric between digitized images," *IEEE transactions on image processing*, vol. 13, no. 12, pp. 1581–1588, 2004.

[47] P. Buneman, S. Khanna, and W.-C. Tan, "Data provenance: Some basic issues," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 2000, pp. 87–93.

[48] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems." in *USENIX Annual Technical Conference, General Track*, 2006, pp. 43–56.

[49] S. Miles, S. C. Wong, W. Fang, P. Groth, K. peter Zauner, and L. Moreau, "Provenance-based validation of e-science experiments," in *ISWC*. Springer-Verlag, 2005, pp. 801–815.

[50] P. Buneman, A. Chapman, and J. Cheney, "Provenance management in curated databases," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 539–550.

[51] S. B. Davidson, J. Crabtree, B. P. Brunk, J. Schug, V. Tannen, G. C. Overton, and C. J. Stoeckert, "K2/kleisli and gus: Experiments in integrated access to genomic data sources," *IBM systems journal*, vol. 40, no. 2, pp. 512–531, 2001.

[52] P. Groth, S. Miles, and L. Moreau, "PReServ: Provenance Recording for Services," in *UK e-Science All Hands Meeting 2005*. EPSRC, 2005.

[53] I. Foster, J. Vockler, M. Wilde, and Y. Zhao, "Chimera: a virtual data system for representing, querying, and automating data derivation," in *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, 2002, pp. 37–46.

[54] P. Townend, P. Groth, and J. Xu, "A Provenance-Aware Weighted Fault Tolerance Scheme for Service-Based Applications," in *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ser. ISORC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 258–266.

[55] J. Cheney, A. Ahmed, and U. A. Acar, "Provenance as dependency analysis," in *Proceedings of the 11th international conference on Database programming languages*, ser. DBPL'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 138–152.

[56] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher *et al.*, "The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud," *Nucleic acids research*, vol. 41, no. W1, pp. W557–W561, 2013.

[57] I. Altintas, O. Barney, and E. Jaeger-Frank, "Provenance collection support in the Kepler Scientific Workflow System," in *Provenance and Annotation of Data*, ser. Lecture Notes in Computer Science, L. Moreau and I. Foster, Eds. Springer Berlin / Heidelberg, 2006, vol. 4145, pp. 118–132.

APPENDIX

A. Use Cases

In this section, we illustrate three different use cases that motivated the design of DAC-MAN.

Data Release Changes in Cosmology. The Sloan Digital Sky Survey (SDSS) [2] collects imaging data of the sky through the optical telescope at Apache Point Observatory in New Mexico, United States. The SDSS data team produces different data releases over time that is accessible to users through an archive server. Each data release consists of a large directory tree with millions of files and terabytes of data. Additionally, the directory tree can be extremely complex due to the existence of symbolic links and several levels of subdirectories. Changes between the data releases often include renaming of files, change of file paths, and changing the data within the files. Users have to identify and understand the different types of changes between these data releases for making informed decisions on data reprocessing and analysis.

Measurement Changes in Environmental Sciences. Fluxnet [1] provides an integrated, processed dataset from over 400 independent flux tower sites that environment scientists can use to analyze the impact of complex biophysical systems on the environmental changes. Currently, these datasets are processed only every few years due to the complexity and amount of data. Each processing cycle results in a different data release, containing an updated version of the previous release along with new data for additional years. The last two data releases, La Thuile (which contains data up to 2007) and Fluxnet2015 (which contains data up to 2015), have changes in column names, floating-point precisions, and file naming conventions, to name a few. Some of these changes may affect only the metadata associated with the datasets (changes in column names, file naming conventions), and the underlying measurement data may still be the same.

Data Management on Supercomputers. Scientific workflows require large amounts of data to be stored and copied/moved between the different tiers of a multi-tiered storage system in an HPC environment. Effective and efficient management of HPC resources for data-intensive computing will require us to make intelligent decisions on when, how and what of the data should be stored and moved. The lack of tools to detect data changes results in multiple copies of the data to co-exist and unnecessary data movement.

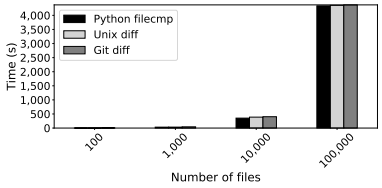


Fig. 14: Scaling issues while detecting changes in large datasets at NERSC. Existing tools like Unix diff, Git and Python filecmp library perform poorly as the number of files in the dataset increases.

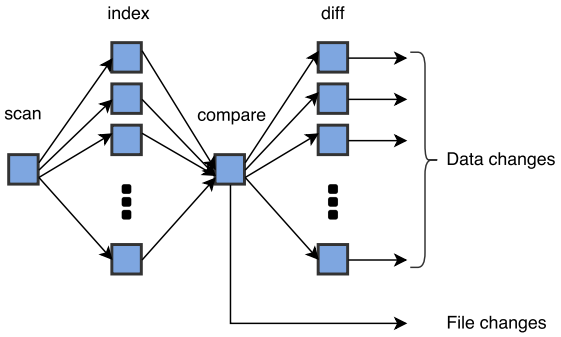


Fig. 15: Change capture workflow in DAC-MAN. DAC-MAN scans and indexes the data for implementing a fast and portable solution to identify and retrieve changes.

B. Scalability of Existing Tools

Figure 14 shows the scalability issue of some widely used tools with increasing amounts of data. The graph shows the time to identify changes using existing tools on a single node at NERSC. The performance deteriorates significantly as the number of files increase.

C. Change Capture Workflow

Figure 15 shows the change capture workflow that is used by DAC-MAN to identify, track and retrieve changes. The change capture workflow is presented as a way for users to configure and setup DAC-MAN based on the data characteristics and needs of the users.

There are four steps in the workflow. The first step (i.e., Scan) crawls the data directories, saving the directory structures and associated file metadata. The second step indexes the files in parallel. The third step uses the indexes to compare the files in the two data directories and saves the changes in a cache, such that it can be reused. The fourth and final step retrieves the changes when a user executes a diff command or could be a summary of all changes in the datasets that is generated automatically.

The parallel indexing step allows DAC-MAN to do a 'fast' comparison. This is because instead of comparing the data for each file, it just uses the indexes to determine the type of change. The type of change can either be *structural*, where only the metadata of a file changes but the contents remain the same, or *semantic*, where the actual values and meaning of data change between the two versions of a file.

The change capture workflow provides flexibility to the users for keeping track of file and data changes. Users can

Change type	Path changed	Name changed	Data changed	Description
None/Unchanged	No	No	No	File metadata (like file path and file name), and data are unchanged between two files
Modified	No	No	Yes	Data changes, but one or more file properties remain the same allowing DAC-MAN to compare files based on names and/or paths
	Yes	No	Yes	
Metadata-only	No	Yes	No	Data remains same, but one or more file properties differ requiring DAC-MAN to find corresponding file names and/or file paths for identifying metadata-only changes
	Yes	No	No	
	Yes	Yes	No	
Added/Deleted	Yes	Yes	Yes	Neither data nor file properties/metdata can be compared

TABLE II: Rules for identifying types of change. DAC-MAN uses only files that are modified for data change analysis. Any other type of change excludes the files from further analysis unless explicitly specified by the user.

Dataset	Files (V_0)	Files (V_1)	Added	Deleted	Modified	Metadata	Unchanged
Synthetic	0 . . . 1,000,000	10 . . . 1,000,000	10 . . . 100,000	10 . . . 100,000	10 . . . 100,000	10 . . . 100,000	0
Redux	4,440,895	5,257,718	827,345	10,522	4,424,011	0	6,362
Resolve	2,254,513	2,254,528	26	11	0	0	2,254,502
Sweeps	31,630	9,865	0	21,765	0	9,865	0
Fluxnet	673	1,629	1,629	673	0	0	0

TABLE III: Different datasets evaluated by the DAC-MAN framework. Based on the different performance metrics, we induce different types and amounts of changes between versions of the synthetic dataset. Redux has versions 5.6.5 and 5.7.0 of the SDSS redux dataset from data release 11 and 12. Resolve contains two versions of the SDSS data release 13 generated using resolve algorithms. Sweeps dataset contains two versions of data from the reduced sweep imaging catalog data files in SDSS. Fluxnet dataset compares 'Fluxnet2015' and 'La Thuile' data releases.

invoke separate steps of the workflow explicitly depending upon the data/directory properties. For example, directories where files are created and deleted very frequently, users can execute the scan and index steps frequently. This is because when the files in the directories are updated, then the previously computed changes become stale. The change capture workflow is then used to rebuild all the indexes and recompute the changes. Directories with WORM (write-once read-many) property need to index the directories and compute changes only once. Since the computed changes should never be stale for such datasets, users can directly retrieve the changes from the cache. Hence, depending on the type of dataset, the different steps of the change capture workflow can be used to run as separate services for identifying, tracking and retrieving changes.

The change retrieval step in the workflow allows users to retrieve changes at different granularities. If the user queries for file changes, the result returns the differences with respect to files in the two directories. If the user queries for data changes, this step uses the file comparison results to calculate data changes between the modified files from the two directories.

D. Classification of Changes

Table II summarizes the different types of changes and associated rules for classifying the files into each type. These rules are based on the changes in directory paths, file names and data changes within the files.

E. Evaluation Dataset

Table III lists the different characteristics of the datasets used in evaluating DAC-MAN.