

FileMap: Map-Reduce Program Execution on Loosely-Coupled Distributed Systems

Michael E. Fisk

Los Alamos National Laboratory
mfisk@lanl.gov

Curtis L. Hash Jr.

Los Alamos National Laboratory
chash@lanl.gov

In this paper we present *FileMap*, an open-source,¹ alternative map-reduce-based computing system that we have developed and utilized over the last 5 years. This system features several significant design decisions and performance aspects that are not found in prevalent map-reduce systems such as Hadoop [16]. The prevailing design goal is to have a system for *scheduling and orchestrating* parallel and distributed data processing, but that does not interpose itself between data and the serial programs that process data. *FileMap* manages the organization of input and output files and the scheduling of program execution, but does not process files itself and is agnostic to the format in which data is stored and/or indexed. We layer on top of existing, ubiquitous file systems, security models, and network access software in order to minimize the complexity of *FileMap* and maximize the ability of its users to benefit from specialized compute platforms, file systems, and software.

We measure the performance of *FileMap* in several instantiations including a heterogeneous “cloud” conglomeration of computers and storage distributed across multiple owning organizations with no cross-organization trust or synchronization. This “cloud” model intentionally supports distributed sensor systems in which nodes collect their own data and participate in analysis of data by moving map/reduce processing upstream to where the data is collected.

Our on SMP systems, clusters built for Hadoop, and this distributed cloud, show that *FileMap* outperforms more prevalent computing systems and models by factors between 2x (compared to Hadoop) and 14x (cloud vs. centralized).

¹<http://mfisk.github.io/filemap>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CloudDP '14, April 13, 2014, Amsterdam, Netherlands.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2714-5/14/04...\$15.00.
<http://dx.doi.org/10.1145/592784.2592790>

FileMap provides a single programming model that allows processing to seamlessly scale from a single laptop to data-intensive compute clusters to distributed sensing and analysis clouds.

This paper introduces our fundamental design decisions in Section 1 and specifics of the implementation in Section 2. Section 3 describes the use of embedded databases within our system. Section 4 presents a performance benchmark measured across a variety of computing environments.

1. Design Decisions and Related Work

While Google was first to popularize the distributed Map-Reduce programming model [4], functional programming with *map* and *reduce* functions dates back at least to the mid 1980's [3]. *Hadoop* [16] is an open-source implementation of distributed Map-Reduce and has become the canonical implementation for many users. However, it has been shown in our tests and in others that Hadoop can often yield performance that falls well short of the optimum [11, 19].

There are other systems for file-based distributed computing, but these previous systems, unlike *FileMap*, do not manage distributed data placement or scheduling of execution in-situ where the files are stored. For example, Condor [13] is a system for running non-interactive programs on loosely-coupled collections of computers, but all file access is remote through either a global filesystem or remote system calls across the network. Makeflow [1] is a file-based data-intensive computing workflow system which utilizes Condor by explicitly copying input and output files across the network or other systems like Hadoop that use a global file system.

The remainder of this section enumerates several other design choices and objectives.

Externalize data ingestion. In contrast to other systems, we impose no overhead or process to ingest data into the system. Data can be ingested through any mechanism that results in it appearing in the file systems used by our system. Sensors can acquire data and drop it directly into local filesystems. Equally, cluster nodes can use parallel filesystems, streaming systems, or offline media to acquire data. The system does not require any notification that additional

data has been added. Continuous or future jobs that apply to the new data will automatically process it. Atomic file modification (typically using a temporary file followed by an atomic filesystem rename) is used to add or update data in the system atomically.

Amortize costs over large chunks of data. Our system does not operate at the level of individual tuples or relations. Instead, we process large batches of data stored in files. A file may contain a single, very complex object or, more typically, millions of individual tuples or relations. We do not optimize the system for small changes to these chunks of data; any change to an input file will result in the complete reprocessing of that file and any downstream processing.

Schedule disks first and then CPUs. In many data-intensive workloads, I/O, rather than CPU, can be the more severe bottleneck. Our scheduler manages both resources but treats filesystems as the fundamental scheduling unit rather than treating a computer or a core as the fundamental unit². The scheduler avoids expensive disk thrashing by only scheduling more than one process per filesystem when the process is CPU bound rather than I/O bound. If multiple users run FileMap concurrently, then each node will be executing at least one process per user. Schedulers use a feedback control mechanism to fairly compete for disk and CPU resources with the objective of not overloading either resource while continuing to make some progress for each user concurrently. On clusters that are intended to give exclusive resource access to one user at a time, we use the SLURM resource manager and batch scheduling [18].

Do not impose a programming language or data storage format. We have designed our system so that it can be used to apply arbitrary commands to arbitrary file types. Linguistic processing can be run on text files. Image processing can be run on images. Network packet processing can be applied to *libpcap* files. In each case, existing, domain-specific tools can be used. Embedded databases are used when more traditional tuple-based storage and SQL queries are desired. Files are processed by map and reduce *programs* rather than by *functions* as in systems like Hadoop. Hadoop’s “Streaming” mode for piping data through programs is widely used but suffers additional performance overhead since the Hadoop runtime interposes itself as a parser and broker between data and the programs. In contrast, FileMap is a scheduling and orchestration system that executes programs directly on stored data.

Zero installation. Computing users often do not have privileged access on either high-performance computing clusters or distributed environments across organizational boundaries. FileMap requires no installation or privileges and carries itself and its configuration to all nodes involved in a computation.

Loose coupling. FileMap supports loosely-coupled clusters, distributed sensors, and networks of workstations. The fundamental connectivity that is required between nodes is remote command execution as provided by *ssh*, *rsh*, or *srml*[18]. Synchronization of work across replicated data is optional. There are no explicit head nodes or lock servers required. Job execution is launched from any system — one of the cluster nodes, a front-end node, or a user workstation.

Do not impose a security model or attack surface and allow existing system security models to be used. The Unix-based computing world has a well-defined set of interfaces, conventions, and functionality. We reuse as many of these as possible in order to improve the flexibility of our system. Specifically, our system can operate on any filesystem implementation and uses Unix users for access control. There are no privileged or supervisor processes in our system. The system can be instantiated with unencrypted, host-based trust implementations like *rsh* and permissive file permissions, or with strongly authenticated and encrypted remote command invocation, fine-grained filesystem ACLs, mandatory access control separations, and strict process accounting. Similarly, competitive scheduling and resource management is also left to the underlying operating system. Our system controls the level of demand placed on the underlying system by an individual user, but relies on the underlying system to schedule processes between competing users.

Scale up and down. The system operates efficiently in single-computer environments as well as larger clusters. When running on a single computer, the system should impose no substantial performance penalty compared to running as a traditional serial pipeline.

Provenance and caching. For scientific reproducibility and to make use of automatic caching of partial results, the system manages input and output locations in a way that implicitly captures the precise derivation of data processing pipelines. It is easy to trace back to the source of any data processing output in the system.

1.1 Inspiration and Related Work

Data-flow through files with automatic reprocessing when input files change timestamps is inspired by *make* [5]. Processing a list of files provided on *stdin* is inspired by *xargs*. The principle that everything can and should be done with remote command execution is inspired by Plan 9 [10]. Decomposition of large data sets with map-shuffle-reduce programming steps is inspired by Google [4]. In short, one could describe our system as distributed parallel *make* with map-reduce operations.

2. Implementation

Users typically submit map/reduce pipelines that specify multiple commands. For example, consider the following

² A computer with multiple filesystems on separate disks (or on separate RAID sets) is actually treated as multiple nodes in the current implementation

FileMap pipeline to compute word frequency across a text corpus:

```
txt2words | sort | uniq -c | partition -f2 <>- total
```

where *sort* and *uniq* are standard Unix commands, *txt2words* tokenizes an input text file to a newline-delimited sequence of words, and *partition -f2* takes an input file, hashes on the second column (the individual word in this case) and creates a number of output files containing partitions of the input file, and *total* which takes count data as produced by *uniq -c* and prints cumulative totals for each word.

The `|` operator has the same pipe semantics as in most Unix shells. However, the `<`, `>`, and `||` operators have different semantics and punctuate where a map/reduce pipeline is divided into *jobs*. We define the semantics of those operators here, followed by the details of how they are executed.

The parallel pipeline operator `||` is defined to have similar semantics to a regular shell pipe, but with caching of the intermediate data to files on disk (as described in more detail in the following subsection).

The operator `<` is defined to mean that each output file (including stdout, which is implicitly created) of the previous command will be distributed to some number of other cluster nodes. The destination nodes are chosen by hashing the basename³ of the output file. Thus, if this pipeline is applied to multiple input files that each create a set of output partition files numbered 1–1000, each file named “1” will be sent to the same set of nodes. Each file named “2” will be sent to some other set of nodes, etc. Writing multiple files is the way that commands enable parallel reduce operations to occur. Traditional Unix programs that output a single text-based output file can be partitioned using other utilities including a FileMap command-line helper utility.

The operator `>` is a reduce operator and specifies that the following command will take multiple input files. The `-` modifier further states that the reduce command will take its list of input files on stdin rather than the command line. This is necessary to support very high-degree reduce operations on traditional Unix systems which limit the number or size of command-line arguments. In our example, one invocation of *total* will be run all of the files named “1” on one of the nodes that received all of those files during the `<` operation. A separate invocation of *total* will process all of the files named “2” and so on.

In our *txt2words* example, if there are 50 input files and the *partition* command creates 100 partitions for each input file, then the 5,000 resulting files are copied between nodes in the cluster. For each partition, one node will receive the 50 inputs and run the *total* command to sum those inputs and produce an output file. The result is 100 output files distributed across the nodes in the cluster.

2.1 Job Execution

The cluster configuration is defined by a single text file. Every user or map/reduce command can use a different cluster configuration or a default configuration file on the system will be used.

Job submission occurs via command line and results in one or more *jobfiles* being placed on each node. Each element of the submitted pipeline maps to a separate job file.⁴ Once those job files are in place, a per-user scheduler process is invoked on each node. If an existing scheduler process is already running for that user, that scheduler will automatically discover the new jobs and begin execution. When there is no more work to be done, the scheduler process exits.

Each jobfile defines a set of input files or file globs to process and a single command (or a pipeline of single-`|` commands) to apply each input. Derived data is stored in a directory tree associated with the input file. For example, if an input file */epubs/alice.txt* is an input to the above pipeline, then a */epubs/alice.txt.d/txt2words* directory will be created⁵ and that will be the current directory when *txt2words* is run. The stdout from the *sort* process will be stored in a file called *stdout* and the stderr (if any) will be stored in a file called *.stderr* (these files are removed upon completion if empty). Runtime statistics including exit status and performance counters are stored in a *.status* file created upon completion of the command. The input file is passed as a command-line argument to the given job command (by default appended to the first or only element of a command pipeline or alternatively substituted for any occurrence of the string *%(input)*). The command may create other files in that directory and, in fact, should create multiple files when serving as a partition operator leading to a shuffle and reduce.

With the exception of reduce steps, the next job in a pipeline is run separately on each output file (that does not begin with a “.” character) of a previous stage. Often there is only one output — the *stdout* file.

The scheduler is designed so that it can be uncleanly killed and restarted at any time. When a job is run on an input file, the output is stored in a temporary directory whose name begins with a “.”. Only when that processing completes is the “.” removed so that the results become available to other jobs. If a job is rerun or the scheduler restarted these temporary directories will be removed and that element of processing restarted.

2.2 Data Transfer Between Nodes

Our system provides *get* and *store* commands to copy data to or from the cluster and *ls*, *mv*, *rm*, and *df* commands to

³ A unix basename is the final component in the `/`-delimited path

⁴ Jobfiles are named with a hash of their contents so that duplicate submissions or resubmissions do not create duplicate jobfiles.

⁵ Non-alphanumeric characters in the command specification are converted to quoted-printable representation to avoid illegal and confusing characters in the file system. Overly-log filenames are truncated and terminated with a hash of the full command line.

examine the aggregate filesystem view comprised of all of the participating nodes.

All file transport between nodes (for jobfiles, redistribute operations, and explicit *get*, *store*, and *replicate* commands) is performed with *rsync* [14] run via remote command execution. When upstream inputs change, downstream computations and redistributes are performed again as well. Use of *rsync* allows us to avoid retransmitting data that has not changed and to intelligently compress data that is not already compressed.

2.3 Synchronization

Synchronization of work being performed across nodes is optional and unnecessary when data is not replicated. However, in a contemporary-style map/reduce cluster where reliability comes from cross-node replication rather than RAID within nodes, there is no point in having redundant computation of the same downstream result. Thus, in this mode of use, a Synchronizer is used to prevent a node from starting an item of work that has already been processed by another node (if that node is still marked as part of the computation and active). Inputs are processed in random order (and replication typically results in no two nodes having the same set of inputs anyway) to minimize the probability that two nodes will even attempt to process the same file at the same time.

The current synchronizer implementation uses a common network filesystem across nodes. Synchronization data is not large, but can have a large number of small requests. The throughput and capacity of this filesystem are not stressed, but file server operations per second is the limiting factor. This has not been a constraint to date but likely would be in larger clusters. Alternate synchronizers could use a web-services API or traditional transactional database for synchronization.

Environments with high-reliability nodes or nodes that collect their own data and do not (or cannot afford to) replicate that data need no synchronization. A distributed sensor environment, for example, would not require synchronization across a wide-area network and would therefore not be sensitive to the extra latency.

Synchronization can be specified per node so that a single system with multiple scheduled filesystems could synchronize with itself using just a local filesystem for synchronization. A distributed cluster comprised of multiple nodes per location with only local replication can be configured to synchronize within a location, but not across locations.

2.4 Inactive Nodes

In large clusters, there will be some nodes down at any time. Our cluster configurations allow nodes to be explicitly declared inactive. When submitting a jobfile to nodes, any non-responsive nodes are marked inactive for the remainder of the job.

When picking nodes to transfer data to for redistribute operations, inactive nodes are skipped and data is guaranteed to be stored onto an active node.

The *store* operation assumes that inter-cluster bandwidth is at least as great as the bandwidth from the storing machine to the cluster, so a single copy of each file is stored into the cluster and then data is replicated within the cluster. It is presumed that important data will be re-replicated periodically so that nodes that have (re-)joined the cluster will receive their share of the data.

2.5 Error Handling

In a research or ad-hoc query environment, it is not uncommon for a user to submit a workload that includes a job which will always fail. For example, the job may be a script or command that is being developed and has a fatal bug. The user may have passed it an illegal set of arguments that result in error. We have learned from experience that computing 100,000 errors before alerting the user is wasteful of both compute and user time. We therefore default to submitting jobs that check the exit status of each process. If any process returns non-zero, that job is terminated on that node. This will usually result in the entire workload terminating as soon as it has no other non-error work to perform. The user is notified as soon as one of the nodes terminates its workload. Exit status is also checked when starting a workload and any cached input that had an error is removed and re-attempted.

Exit status can be optionally ignored when it is permissible for some commands to return an error, but this is not recommended.

2.6 Replication

We use a consistent hashing algorithm [12] to pick the nodes that should contain replicas of data or process specific partitions of data. Consistent hashing provides a statistically even distribution of data while minimizing the amount of data that needs to be relocated when a node is added or removed. The fraction of data that is assigned to a node can be scaled manually to deal with nodes that have disproportionately small or large amounts of storage.

3. Query Functionality

We have designed our system so that it can be used to apply arbitrary commands to arbitrary file types. Linguistic processing can be run on text files. Image processing can be run on images. Network packet processing can be applied to *libpcap* files. In each case, existing, domain-specific tools can be used.

However, it is also common to process tabular data and to want to perform map operations that are standard selection, projection, and/or aggregation operations on that data. Reduce operations include the bag-union and relational queries. Thus, without limiting our system to a particular database implementation, we do seek to provide a stylized, but familiar query environment to users. Automatic query planning of

a complex query as a map/reduce pipeline has been done by others in [2, 8] and would be advantageous as well. Our previous work on the System for Modular Analysis and Continuous Queries [6, 7] performs algebraic rewriting of multiple concurrent stream queries.

We have therefore implemented a *db* command that uses embedded database libraries to manage local storage and query of tabular data. By using filesystem-based embedded databases, we can use our system’s models for replication, communication, and scheduling while inheriting the index and/or query functionality of the embedded database. Our implementation currently supports *sqlite3* [9], HDF, *Fastbit* [17], CSV, and column-oriented text databases. Each of these embedded databases uses local files for storage rather than relying on a network service to serialize access, as is the case for MySQL, Oracle, DB and other multi-user databases. Wherever possible we explicitly disable transactions and journaling to improve performance since the basic job scheduling mechanisms of our system already prevent readers from accessing a database that is still being generated.

Our *db* command allows our previous map/reduce workload to be rewritten with the following database commands rather than requiring that *partition* and *total* text processing tools be created:

```
txt2words
| db -import word:text
|| db -q "count(*), word" -k word -n 256
<>- db -q "sum(count), word"
```

The *db* command always creates a new output database from one or more input databases. The *import* option creates a new table from tabular text input. When a *-q* option is given, the specified query is applied to the input(s). When the *-k* options is given, the specified field(s) are used to partition the input across *-n* output databases. Note that these commands assume a 1-1 relationship between databases and tables so that table names are unnecessary. We use the implicit-groupby shorthand where invoking an aggregate function like *sum()* implicitly causes a *group by* operation on all non-aggregate fields.

Our *db* command uses a library for processing tabular data with SQL queries. Queries are parsed into three elements: read columns, select (filter), and project (including aggregates). Data providers can implement simple read, read with select, or full SQL (read with select and project). The library provides default in-memory implementations of the select and project for data sources that do not provide them.

4. Performance

In this section we quantitatively show that FileMap allows for a single programming model to be automatically scaled on different platforms: a laptop, SMP, cluster, and a geographically distributed, heterogeneous cloud.

Platform	Baseline	Time (secs)	FileMap	
			Time	Speedup
Laptop	Serial	315.7	410.6	0.8x
Laptop (gzip)	Serial	227.4	378.5	0.6x
SMP	Serial	246.3	236.4	1x
SMP (gzip)	Serial	236.4	24.3	10x
Cluster	Hadoop	104.0	52.5	2x
Cloud	Get + SMP	403.2	29.3	14x
Cloud (gzip)	Get + SMP	348.9	35.4	10x

Table 1. Performance comparison on multiple platforms

We conducted benchmarks with a public dataset for reproducibility. Specifically, we used the Daily Global Historical Climatology Network dataset [15] which consists of 86,670,524 records and is 22GB in uncompressed form. For our tests, we split the dataset into 100 evenly-sized files.

The analysis performed was a histogram of weather station observations (histograms are a common, simple benchmark for map-reduce systems). All processing was done with the *cut* and *awk* commands and *zcat* was used in the cases where data was compressed.

We conducted tests on 4 platforms: a laptop, a 48-way SMP, a cluster built for Hadoop, and a geographically distributed heterogeneous cloud. We measured baseline serial performance on the laptop and SMP. We measured FileMap performance on all platforms. Hadoop performance was measured on the Hadoop cluster.

4.1 Cloud Configuration

The cloud used in this paper consisted of 63 computers in 5 different U.S. cities including 3 separate small clusters, 2 virtual machines at a commercial cloud hosting service, and a network of workstations in a computer lab. While storage space is not the focus of this paper, the cluster totals 216 TB of capacity.

No software was installed on any of these systems for these tests and no privileged-user intervention or configuration was required other than creating an account for the author (with the obvious exception that Hadoop was pre-installed on the Hadoop cluster). The systems are administered by 6 different owning organizations. There are no network filesystems that are cross-mounted between owning organizations and the organizations were unaware of the other organizations involved.

This cloud configuration consists only of Linux computers, but is otherwise relatively heterogeneous. It runs 12 different Linux kernels and 9 different versions of 4 Linux distributions. Hardware consists of 184 hardware threads of 14 different CPU types including Intel Xeon, AMD, and Intel Core i5. Three different filesystem types are used. FileMap is written in Python and the cloud has 5 different versions of Python.

The baseline test on our distributed cloud computation, which was designed to mimic a distributed sensor system,

was to copy all data from the distributed sensors to the SMP and then analyze the data there. This copy was done with parallel *rsyncs* from the SMP to each of the other computers. The load average during this copy peaked at 3.5 and the average network utilization was less than 150 Mb/s although with compression the effective transfer rate of uncompressed data was 1.2 Gb/s.

4.2 Results

Table 1 summarizes the results. FileMap outperforms the alternative on all multi-node tests including a 2x improvement over Hadoop and 10-14x improvements in the cloud test.

On the SMP, FileMap did not exceed serial performance with uncompressed data since disk I/O was the bottleneck even in the serial case. However, when the data is compressed, FileMap parallelization on the SMP results in a 10x speed improvement.

In the laptop case, there is an overall loss in performance as a result of a single serial program nearly completely consuming the laptop's resources; since the first program is not quite utilizing the machine, FileMap starts a second concurrent process which happens to result in thrashing behavior. Future scheduler improvements could resolve this performance issue in environments where parallelism is not desirable.

5. Conclusion

We presented a novel approach to map-reduce system design and implementation and demonstrated the effectiveness of this approach with benchmarks on a spectrum of platforms. The use of geographically distributed cloud of loosely-couple machines is particularly and uniquely strong for FileMap compared to other systems that assume that computations are performed within a single administrative domain.

We have described both a specific system instantiation and its performance, but also the design decisions that drove system development which may inform other system designs.

References

- [1] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: a portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, page 1. ACM, 2012.
- [2] S. Babu. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 137–142. ACM, 2010.
- [3] R. Bird. Lectures on constructive functional programming. *Constructive Methods in Computer Science*. Springer-Verlag. Also available as *Technical Monograph PRG-69*, from the *Programming Research Group, Oxford University*, 1988.
- [4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [5] S. I. Feldman. Makea program for maintaining computer programs. *Software: Practice and experience*, 9(4):255–265, 1979.
- [6] M. Fisk. System for modular analysis and continuous queries, 2003. URL <http://smacq.sf.net>.
- [7] M. Fisk and G. Varghese. Agile and scalable analysis of network events. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement*, pages 285–290. ACM, 2002.
- [8] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX Annual Technical Conference*, pages 267–273, 2008.
- [9] M. Owens. *The definitive guide to SQLite*. Apress, 2006.
- [10] R. Pike, D. Presotto, K. Thompson, H. Trickey, et al. Plan 9 from bell labs. In *Proceedings of the summer 1990 UKUUG Conference*, pages 1–9, 1990.
- [11] J. Shafer, S. Rixner, and A. L. Cox. The hadoop distributed filesystem: Balancing portability and performance. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 122–133. IEEE, 2010.
- [12] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM Computer Communication Review*, volume 31, pages 149–160. ACM, 2001.
- [13] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.
- [14] A. Tridgell. *Efficient algorithms for sorting and synchronization*. Australian National University Canberra, 1999.
- [15] U.S. National Oceanic and Atmospheric Administration. Daily global historical climatology network-daily (version 3.11), Sept. 2013. URL <ftp://ftp.ncdc.noaa.gov/pub/data/ghcn/daily/>.
- [16] T. White. *Hadoop: the definitive guide*. O'Reilly, 2012.
- [17] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, et al. Fastbit: interactively searching massive data. In *Journal of Physics: Conference Series*, volume 180, page 012053. IOP Publishing, 2009.
- [18] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.
- [19] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.