Astronomy in the Cloud: Using MapReduce for Image Co-Addition

K. WILEY, A. CONNOLLY, J. GARDNER AND S. KRUGHOFF

Survey Science Group, Department of Astronomy, University of Washington; kbwiley@astro.washington.edu

AND

M. BALAZINSKA, B. HOWE, Y. KWON AND Y. BU

Database Group, Department of Computer Science, University of Washington

ABSTRACT. In the coming decade, astronomical surveys of the sky will generate tens of terabytes of images and detect hundreds of millions of sources every night. The study of these sources will involve computation challenges such as anomaly detection and classification and moving-object tracking. Since such studies benefit from the highest-quality data, methods such as image co-addition, *i.e.*, astrometric registration followed by per-pixel summation, will be a critical preprocessing step prior to scientific investigation. With a requirement that these images be analyzed on a nightly basis to identify moving sources such as potentially hazardous asteroids or transient objects such as supernovae, these data streams present many computational challenges. Given the quantity of data involved, the computational load of these problems can only be addressed by distributing the workload over a large number of nodes. However, the high data throughput demanded by these applications may present scalability challenges for certain storage architectures. One scalable data-processing method that has emerged in recent years is MapReduce, and in this article we focus on its popular open-source implementation called *Hadoop*. In the Hadoop framework, the data are partitioned among storage attached directly to worker nodes, and the processing workload is scheduled in parallel on the nodes that contain the required input data. A further motivation for using Hadoop is that it allows us to exploit cloud-computing resources: *i.e.*, platforms where Hadoop is offered as a service. We report on our experience of implementing a scalable image-processing pipeline for the SDSS imaging database using Hadoop. This multiterabyte imaging data set provides a good testbed for algorithm development, since its scope and structure approximate future surveys. First, we describe MapReduce and how we adapted image co-addition to the Map-Reduce framework. Then we describe a number of optimizations to our basic approach and report experimental results comparing their performance.

1. INTRODUCTION

Many topics within astronomy require data gathered at the limits of detection of current telescopes. One such limit is that of the lowest detectable photon flux given the noise floor of the optics and camera. This limit determines the level below which faint objects (*e.g.*, galaxies, nebulae, and stars), cannot be detected via single exposures. Consequently, increasing the signal-to-noise ratio (S/N) of such image data is a critical data-processing step. Combining multiple brief images, *i.e.*, *co-addition*, can alleviate this problem by increasing the dynamic range and getting better control over the point-spread function (PSF). Co-addition has successfully been used to increase S/N for the study of many topics in astronomy, including gravitational lensing, the nature of dark matter, and the formation of the large structure of the universe. For example, it was used to produce the Hubble Deep Field (Szalay 1999).

While image co-addition is a fairly straightforward process, it can be costly when performed on a large data set, as is often the case with astronomical data (*e.g.*, tens of terabytes for recent surveys and tens of petabytes for future surveys). Seemingly simple routines such as co-addition can therefore be a time- and data-intensive procedure. Consequently, distributing the workload in parallel is often necessary to reduce the time to solution. We wanted a parallelization strategy that minimized development time, was highly scalable, and was able to leverage increasingly prevalent service-based resource-delivery models, often referred to as *cloud computing*. The MapReduce framework (Google 2004) offers a simplified programming model for data-intensive tasks and has been shown to achieve extreme scalability. We focus on Yahoo's popular open-source Map-Reduce implementation called Hadoop.[1] The advantage of using Hadoop is that it can be deployed on local commodity hardware but is also available via cloud resource providers (an example of the "*platform as a service*" cloud-computing archetype). Consequently, one can seamlessly transition one's application

---

[1] See http://hadoop.apache.org/.

between small- and large-scale parallelization and from running locally to running in the computational cloud.

This article discusses image co-addition in the cloud using Hadoop from the perspective of the *Sloan Digital Sky Survey* (SDSS) with foreseeable applications to next-generation sky surveys such as the *Large Synoptic Survey Telescope* (LSST).

We recognize that some of the terms used in this article are employed by the image-processing and astronomical communities in occasionally ambiguous ways. To clarify our exposition we have chosen the following definitions, not because they are necessarily the only acceptable definitions, but rather so that a requisite consistency can be imposed on our description. The term *co-addition* refers to the overarching process of combining a set of images into a final image. No assumptions are made about the input set. For example, they may have undergone no initial processing such as background subtraction or flat-field correction, they may only partially overlap, they may reside in different coordinate systems or in different projections of a given coordinate system, or they may have varying point-spread functions. The term *stack* refers to a set of images that have been fully registered in the ways just listed but that have not been flattened yet. The images residing within a stack have undergone all initial processing (background subtraction, etc.), have been projected (and resampled) to a common coordinate system, and have been matched to a common point-spread function. The term *mosaic* refers to the final product of the co-addition process; *i.e.*, a mosaic is a single flattened image of the stack.

In § 2, we present background and existing applications related to the work presented in this article. In § 3, we describe cloud computing, MapReduce, and Hadoop. In § 4, we describe how we adapted image co-addition to the MapReduce framework and present experimental results of various optimization techniques. Finally, in §§ 5 and 6, we discuss final conclusions of this article and describe the directions in which we intend to advance this work in the future.

2. BACKGROUND AND RELATED WORK

2.1. Image Co-Addition

Image co-addition is a well-established process in which a set of images go through a multistage image-processing pipeline to produce a single image of higher quality than the individual images in the input set. The stages of this pipeline are described subsequently. Due to its established methodology and important utility to researchers, many variants of co-addition have been implemented in the past. Examples include *SWarp*[2] (Bertin 2002), the SDSS co-adds of Stripe 82 (SDSS DR7[3]), and *Montage* (Jacob 2009; Montage [4]), a scalable image analysis framework that can be run on parallel systems. SWarp does

support parallel co-addition; it is primarily intended for use on desktop computing environments, and its parallelism is therefore in the form of multithreaded processing. Parallel processing on a far more massive scale is required for existing and future data sets. Fermilab produced a co-added data set from the SDSS database (the same database we processed in this research), but that project is now complete and does not necessarily represent a general-purpose tool that can be extended and reused.

Montage is most closely related to our work but there are notable differences. Montage is a software toolkit and grid portal for constructing co-added and mosaiced images from large data sets. When run in its parallel form on a grid, e.g., on *Tera-Grid*,[5] the concurrent processing and internode communication is organized using a message-passing interface, the standard library for message-passing parallelization on distributed-memory platforms. MapReduce is a higher-level approach that offers potential savings in development time, provided one's application can be projected well onto the MapReduce paradigm. Hadoop, in particular (in congruence with Google's original MapReduce implementation), was also designed to achieve high scalability on inexpensive commodity hardware, rather than requiring parallel machines with high-performance network and storage.

Image co-addition is a method of image processing in which multiple overlapping images are combined into a single image called a *mosaic* (see Figs. 1 and 2). The process is shown in Table 1. First, a set of relevant images is selected from a larger database. To be selected, an image must be in a specified band-pass filter (*line 6*) and must overlap a predefined bounding box on the sky (*line 9*). The image is first *background-subtracted* (or *background-rectified*) to remove spurious background radiation and is then *registered* with the output coordinate system through a process of *image-warping*. Warping assigns a value to each pixel in the output by mapping that pixel's location to a corresponding location in the input image and then interpolating the input image at that location (*line 10*). Finally, the image is then adjusted to a common PSF (*line 10*). The set of all input images processed in this manner comprises a *stack*, *i.e*, a set of registered images. The stack is then summed on a per-pixel basis to produce a single consistent *mosaic*: *i.e.*, a flattened image of the stack (*line 18*).

2.2. Experimental Data Set

Much of modern astronomy is conducted in the form of large sky surveys in which a specific telescope and camera pair are used to produce a consistent[6] database of images. One recent and ongoing example is the *Sloan Digital Sky Survey* (SDSS), which has thus far produced 70 Tbytes of data (SDSS DR7).
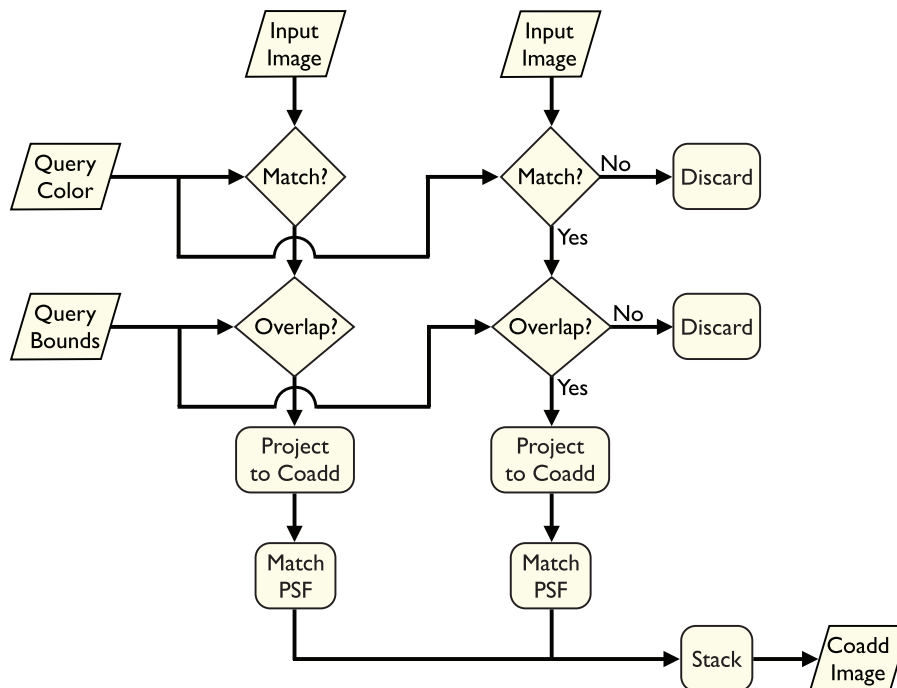
Fig. 1.—Flowchart of the overall image co-addition process. The number of input images is effectively unlimited. Note that multiple independent simultaneous queries are also supported.

The next-generation survey is the LSST, which will produce 60 Pbytes of data over its 10 yr lifetime, starting this decade. These data sets are large enough that they cannot be processed on desktop computers. Rather, large clusters of computers are required.

In this article we consider the SDSS database. The SDSS camera has 30 CCDs of 2048×1489 resolution arranged in a 5×6 grid representing five bandpass filters and six strips of sky (Gunn 1998). Images are captured by drift-scan such that the camera remains fixed during an integration session and the sky moves across the field of view. The CCD readout rate is slowed down dramatically (as compared with a typical rapid-capture CCD) so as to shift pixel charge from one row of the CCD to the next in precisely sidereal time. In this way, images are captured without trails, yet without moving the tele-scope. The resulting integration time per point on the sky is con-sequently determined by the arc subtended by a CCD's length along the right ascension axis relative to the sky (54.1 s).

2.3. Cluster Configuration and Experiment Data Set

We ran our experiments on a Hadoop-configured cluster. This cluster is funded by the National Science Foundation *Clus-ter Exploratory* grant (CluE) and is maintained by Google and IBM. At the time of this research, the cluster consisted of approximately 400 quad-core nodes (more nodes were added later), with each node divided evenly between mapper and re-ducer slots (described in § 3). Thus, there were approximately 800 mapper and reducer slots each. It should be noted that other people working on unrelated projects also have access to this cluster for research purposes and, due to the unpredictability of their usage at any given time, the cluster's load could vary dramatically. This situation introduced wide variances into our timing experiments.

For our experiments we used the Stripe 82 data set from the SDSS-I (2000–2005) and SDSS-II (2005–2008) surveys. The images are minimally altered[7] from the original camera-captured images and therefore reflect the original CCD resolu-tion (described previously) and world coordinate system (WCS). The images undergo a coordinate-system transformation based on the WCS in each FITS file's header as a component of the co-addition process. Stripe 82 spans right ascension between $-50°$ and $+60°$ (Stripe 82 only operated in the fall) and has a coverage of about 75 visits (see Fig. 3). Stripe 82 contains about 1,000,000 images in about 20 Tbytes. From this database we defined a 3° long window in right ascension in the most deeply covered region (37° to 40°; see Fig. 3) and produced the corre-sponding subset for our experiments, approximately 100,000 images comprising 250 Gbytes of gzip-compressed data (about 600 Gbytes uncompressed). Using a subset permits us to perform

---

[7] Bias subtraction, flat-field correction, and cosmic-ray/defected-pixel interpolation (see the SDSS glossary at http://www.sdss.org/dr7/glossary/).

Fig. 2.—Effect of image co-addition. Both images show the same region of sky (SDSS Stripe 82, R bandpass), but the image on the bottom is composed from 79 independent exposures. The S/N is improved by approximately a factor of 9, the observable result of which is a dramatic decrease in the noise (grain), thus revealing numerous sources that are undetectable in the single exposure shown on the top.

our experiments much more rapidly. Note that the subset maintains the full depth of the original data set by being confined to all images within a given spatial bound. A more naive method of generating a subset, perhaps by randomly selecting a fraction of the original data set, would have yielded shallower co-adds, and any experiments would therefore have offered weaker insight into how the system's performance might scale up to the full data set.

We ran the Hadoop co-add system described in this article on two different queries: one representing a lower bound on the query sky bounds we expect to be requested when conducting actual research (approximately $1/4°$ square) and the other representing the corresponding upper bound (approximately $1°$ square). Thus, the two resulting running times demonstrate the expected best and worst performances under real-world conditions.

In addition to the two queries that represent our primary experiments, we also briefly considered other queries of identical size but at different locations within the subset's sky bounds in order to determine the degree to which performance might depend on the specific query. Most notably, the larger query might cover either three or four SDSS strips (camera columns), depending on its precise position along the declination axis. Such a difference could conceivably have an effect on the performance of the system, since it affects the degree to which our prefiltering mechanism (see § 4.1.1.) can eliminate SDSS strips (a filter pass of 4/6 strips in one case and 3/6 strips in the other), thereby imposing a subsequent variation in the number of false positives that must be considered by MapReduce. Furthermore, due to the general variability of the database, two queries of identical size but located in different locations will almost certainly have some variation in their overall coverage: *i.e.*, in the number of FITS images that ultimately contribute to the mosaic. The results of these experiments have suggested that variation in performance resulting from the factors described here is minimal. The final running time of the overall Hadoop job is virtually unaffected by the choice of query location. Without loss of generality, we will therefore dispense with any consideration of such variation for the remainder of the article.

3. MAPREDUCE

MapReduce is a programming model and an associated implementation for parallel data processing (Dean 2004). *Hadoop* (Apache 2007) is the open-source implementation of MapReduce. MapReduce has a few primary motivations underlying its design. First, the overarching goal is to achieve *data locality*, *i.e.*, to move the computation to the data, rather than to move the data to the computation. Specifically, Hadoop attempts to dispatch each task in the distributed computation to the node that holds the data to process. Another goal of the MapReduce design is to present a simplified framework for describing a data-processing pipeline such that most general applications can be described within that framework. Finally, MapReduce is designed to scale to clusters with thousands of machines. At this scale, failures are the norm, rather than the exception. It thus includes machinery to hide compute-node failures from the user program by managing multiple replicas of all data blocks, automatically restarting tasks that fail, and optionally starting multiple redundant tasks to reduce restart overhead when failures occur.

MapReduce consists of two sequential stages in which units of computation called *mappers* and *reducers* process the initial and intermediate representations of the input data, respectively. While these two stages occur sequentially (mappers first), data processing can be highly parallel *within* each of these two stages. In the *map* stage, a large input data set is broken into smaller pieces and distributed to numerous independent mapper objects. Each map processes its input data, in effect performing a transformation of that data to an intermediate representation.

TABLE 1
ALGORITHM 1: CO-ADD

| |
| --- |
| **10** Input: {ims: one set of images, qr: one query} |
| Output: {mosaic: a single co-added image, depth: mosaic's per-pixel coverage map} |
| 1:   twarps ← empty // set of warped intersections |
| 2:   qf ← filter // bandpass filter |
| 3:   qb ← qr bounds |
| 4:   For each image im ∈ ims, do |
| 5:   imf ← im filter |
| 6:   If imf=qf, then |
| 7:   imb ← im bounds |
| 8:   twc ← intersection of qb and imb // Get intersection in R.A./decl. |
| 9:   If twc is not empty, then |
| 10:   twarp ← warping of im to twc // Astrometry/interpolation/PSF-matching |
| 11:   Add twarp to twarps |
| 12:   End if |
| 12:   End if |
| 14:   End for |
| 15:   mosaic ← initialized image from qr |
| 16:   depth ← initialized depth map // same dimensions as mosaic |
| 17:   for each image twarp ∈ twarps, do |
| 18:   Accumulate twarp illumination into mosaic |
| 19:   Accumulate twarp coverage into depth |
| 20:   End for |
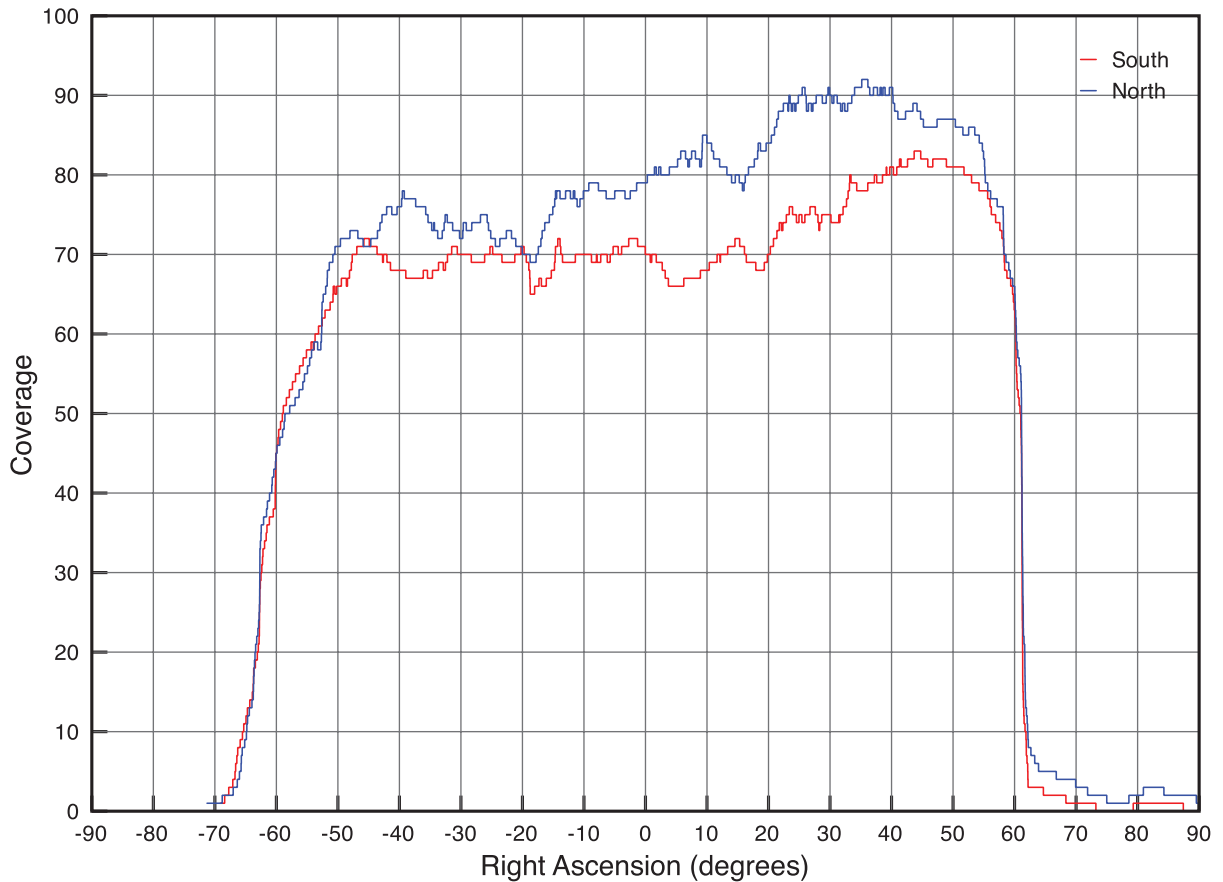
FIG. 3.—SDSS Stripe 82 coverage as a function of R.A. Within its declination range of ±1.25°, average coverage is about 75 between R.A. −50° and +60°.

TABLE 2
ALGORITHM 2: MAP

Input: {init: qr: one query, key: unused, value: im: one image}	
Output: {key: qr, value: twarp: a single image warped to qr bounds}	
1:	imf ← im filter
2:	imb ← im bounds
3:	qf ← qr filter
4:	qb ← qr bounds
5:	If imf=qf, then
6:	twc ← intersection of qb and imb // Get intersection in R.A./decl.
7:	If twc is not empty, then
8:	twarp ← warping of im to twc // Astrometry/interpolation
9:	End if
10:	End if

TABLE 3
ALGORITHM 3: REDUCE

Input: {key: qr, value: twarps: a set of images warped to qr bounds}	
Output: {key: unused, value: unused}	
1:	mosaic: a single co-added image
2:	depth: a coverage map corresponding to mosaic
3:	mosaic ← initialized image from qr
4:	depth ← initialized depth map // Same dimensions as mosaic
5:	For each twarp ∈ twarps, do
6:	Accumulate twarp illumination into mosaic
7:	Accumulate twarp coverage into depth
8:	End for
9:	Write mosaic and depth to FITS files // Side effect of MapReduce process

Following the map stage, the full set of intermediate map outputs is further distributed to the *reduce* stage. This distribution process is basically a massive distributed sort of the mapper outputs, followed by assignment of the outputs to the reducers for subsequent processing. Following the mapper and intermediate shuffle stages, the reducers perform a second transformation, the output of which is the output of the entire MapReduce process. MapReduce may be followed by a final merge stage that groups the numerous reducer outputs into a single result, but this last step is generally a simple concatenation and is therefore not considered an integral step in the overall MapReduce process. While MapReduce can be performed in more complex ways such as running multiple sequential map stages before the final reduce stage (each map stage's outputs feeding the inputs of the next map stage), in this article we concentrate on the simpler case of a single map stage followed by a single reduce stage.

The implementation of MapReduce used in this article is Hadoop. Hadoop is implemented in Java and, in addition to the general MapReduce framework, incorporates the necessary additional components to build a full MapReduce system: *i.e.*, a distributed file system (HDFS), a job scheduler, etc.

4. IMPLEMENTING IMAGE CO-ADDITION IN MAPREDUCE

To adapt a general algorithm to MapReduce we must define two functions: the *mapper* and the *reducer*. We have adapted image co-addition to MapReduce in the most direct manner possible (see Tables 2 and 3). This straightforward implementation suffices to demonstrate the key benefits for the MapReduce framework as a building block for astronomy image-processing pipelines. Figure 4 illustrates the overall process. Each call to the map function processes a single input image. The image is checked against the query parameters for inclusion in the mosaic. If it passes this test (overlap of the query bounds and designation as the correct bandpass filter), it is then warped to the query's coordinate system and the resulting bitmap is passed to the reducer. After all images have been excluded or warped, the reducer then accumulates all of the bitmaps into a mosaic.[8]

Table 3 shows that we use a single reducer to perform the summation of the warped intersection bitmaps for a given query; *i.e.*, while mappers are parallel over input images, reducers are parallel over queries. The overall process can readily be extended to process multiple queries at once.[9] In such cases, the mapper considers the input image against each query and produces a warped bitmap for each query. The bitmaps are then sent to distinct reducers, each of which performs the summation for a particular query. Note that the warping of the input images to the query's coordinate system dominates the computational cost; *i.e.*, the per-pixel summation of the stack that is performed in the reducer is a comparatively simple operation. Furthermore, in our future work we intend to significantly increase the complexity of the mapper operation through the incorporation of more sophisticated warping and PSF-matching algorithms, whereas the reducer's task of pixel summation will remain relatively unaffected. Therefore, the reducer's serial nature does not hinder the overall running time. Finally, it should be noted that while the reducer acts in serial on a per-query basis, it can act in parallel across queries if a single job processes multiple queries against the overall data set (see Fig. 4). While the output of the reducer is formally a key/value pair our system does not emit the mosaic (the image itself) through this mechanism. Rather, the mosaic is written directly to disk (as a FITS file) as a side effect of the MapReduce process.

The naive method described in this section summarizes the most straightforward adaptation of image co-addition to MapReduce. However, it suffers from two inefficiencies. First, the naive method processes every image once every execution. Second, finding the location of millions of small individual files during processing dominates the runtime. In the next section,

---

[8] Note that in our current work, we are not performing PSF-matching, only warping and interpolation.

[9] In fact, we have implemented the processing of multiple queries, but not thoroughly profiled such behavior.
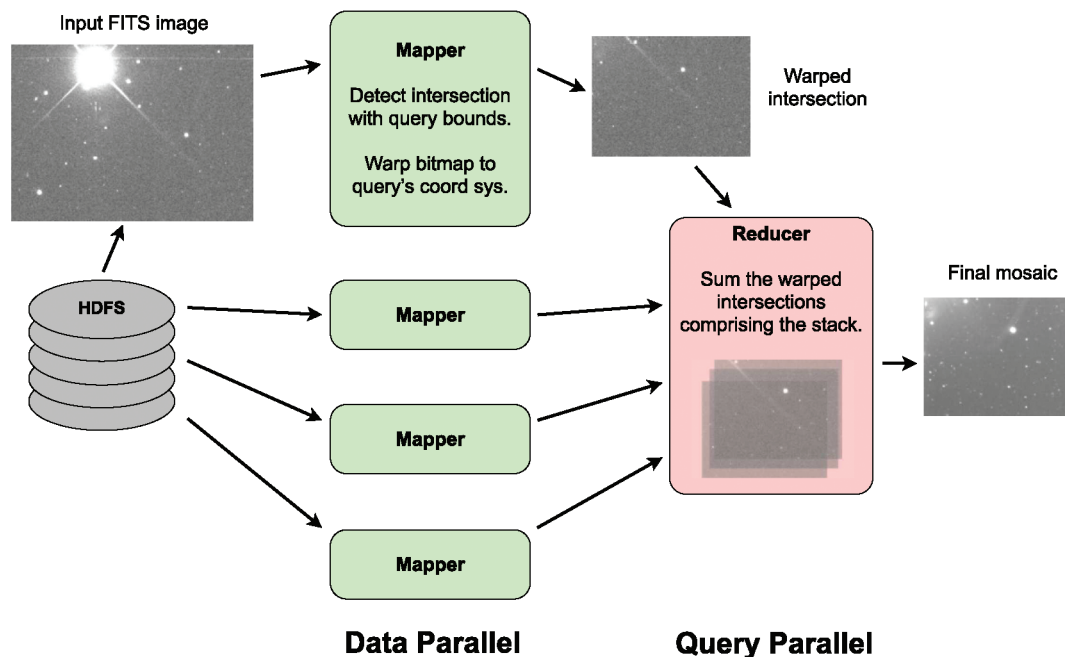
FIG. 4.—Image co-addition adapted to MapReduce. Each input image from the database is processed by a single mapper. Given multiple mappers operating on multiple nodes of the cluster, this stage can be highly parallel. Each mapper sends a warped bitmap (the intersection of an input image and the query bounds) to a reducer. The reducer operates in serial for a given query, accumulating all of the bitmaps into a mosaic. Note, however, that processing multiple queries is quite feasible. In such a scenario, the mappers process their input image against multiple queries and send each corresponding bitmap to a different reducer. In this way, the reducers operate in parallel over the multiple queries of a given run.

we address these problems and describe optimizations to solve them.

4.1. Optimizations

4.1.1. Prefiltering

Of the 100,000 FITS files in our experimental data set, only a small number will ultimately contribute to any one query's mosaic. If we can exclude some of the irrelevant files from consideration before running MapReduce we may improve the overall job's performance. Therefore, one of the most straightforward ways to improve efficiency is to filter the input and not process the entire data set. Filtering can be accomplished by considering the data set's directory structure and file name convention and subsequently building a partially exclusive *glob* pattern (similar to a regular expression) when specifying the input file set for MapReduce.

Our prefilter works in two ways. The SDSS camera's 30-CCD array is divided into five rows corresponding to five filters and six columns corresponding to six parallel nonoverlapping strips of the sky. In the Stripe 82 data set, these six strips correspond to parallel strips in declination. Any individual FITS file in the data set corresponds to an image captured by one of these 30 CCDs. Since a query explicitly indicates a specified bandpass, we can immediately reduce our input data set by a factor of 5 by eliminating from consideration those FITS files

captured in one of the four other bandpass filters. Furthermore, the data set is structured in such a way that is relatively straightforward to filter by CCD column; *i.e.*, by spatial location along the declination axis (see Fig. 5). We perform this secondary filter by determining which of the six columns the query's sky bounds overlap. For a very small query we might therefore achieve a further sixfold reduction in the input data set, although for larger queries this reduction is usually on the order of 1.5 at best (elimination of two of the six total columns).

The following diagram shows how the SDSS Stripe 82 database is organized. As an example, let us assume that we wish to construct a filter that accepts strips 2, 3, and 4, and bandpass g. One possible glob[10] would be /SDSSDB_ROOT/*/*/corr/[234]/fpC-*-[g][234]-*.fit. The diagram below, of the directory hierarchy of the SDSS database, illustrates how this glob pattern can be used to exclude the unnecessary files when building the input set for MapReduce.

```
SDSSDB_ROOT
  5902 ← SDSS Stripe 82 run id
    40 ← rerun id
```

---

[10] There are multiple possible ways of constructing a glob that satisfies the filter requirement. For example, the second [234] in the example shown is redundant and could be replaced with a *.

```
      corr ← constant directory name (cor-
related images)
        1 ← strip (camera column 1-6)
          fpC-005902-u1-0690.fit ← FITS
file
          fpC-005902-g1-0690.fit
          fpC-005902-u1-0691.fit
fpC-005902-g1-0691.fit
        2
          fpC-005902-u2-0690.fit
          fpC-005902-g2-0690.fit
          fpC-005902-u2-0691.fit
          fpC-005902-g2-0691.fit
        3
        4
        5
        6
```

We ran our experiments on two query sizes. Table 4 (*row 2*) and Figure 6 (*first bar in each set*) show the results of our first round of experiments with the larger query's results in the left column of the table and left plot group. Note that the spatial filter occurs on only one spatial axis, not two as would be required for a perfect filter. Given a column that passes the prefilter, all FITS files in the data set that were captured in that column will be considered by the mappers. This is unfortunate, since many of those FITS files will still not overlap the query bounds in right ascension. They will be detected via their empty intersections and be discarded by the mappers, but these false positives represent wasted effort. We discuss this inefficiency and methods for handling it in § 4.1.4., but first we investigate where the prefiltered method focused its time.

Figure 7 shows a breakdown of the prefiltered job's overall running time for the larger query into the salient stages. The last bar represents the total running time, 42 minutes, the same measurement indicated in the first bar of Figure 6. A call to `main()` involves a preliminary setup stage in the *driver* followed by a single call `runJob()`, which encapsulates the MapReduce process. Thus, the bar in the fourth region is the sum of the bars in the first and third regions. Likewise, the call to `runJob()` runs the *MapReduce* process and has been further broken down into the components shown in the second region. Thus, the bar in the third region is the sum of the bars in the second region. We observe that Construct File Splits is the dominating step of the overall process. This indicates that the Hadoop co-add process spent most of its time locating the FITS files on HDFS and preparing them as input to the mappers. The cost associated with this step results from a set of serial remote procedure calls (RPCs) between the client and the cluster. The actual image-processing computations are represented only in the two shortest bars labeled *Mapper Done* and *Reducer Done* indicating that Hadoop spent a small proportion of its total time on the the fundamental task. This observation suggests that there is substantial

FIG. 5.—In addition to prefiltering images by bandpass filter, we also prefilter images on one spatial axis. Given that the SDSS camera produces images in six parallel strips of sky, we can exclude those strips that do not overlap the query bounds. This figure shows the spatial layout of a set of FITS files as produced by the camera, arranged into six long strips. Since the query bounds (*inset rectangle*) only overlap columns 2, 3, and 4, we can exclude columns 1, 5, and 6 (indicated with X). However, the filter suffers from false positives (indicated with FP). Those images will be detected and discarded in the mappers.

inefficiency in the associated method: namely, in the RPCs involved. The next section describes one method for alleviating this problem.

We never explicitly measured the performance without prefiltering, but an estimate is easy to calculate. Given that the running times with prefiltering for our two experimental query sizes were 26 and 42 minutes, respectively, and that the running time was dominated by the serial RPC bottleneck, and that the prefilter reduced the number of input files by a factor of 7.5 (about 13,000 files out of the 100,000 total passed the prefilter), we can estimate that without prefiltering, the process would have taken approximately 194 and 315 minutes, respectively (Table 4, row 1). This estimate assumes a linear relationship between

TABLE 4
Co-Add Running Times (Minutes) for Two Query Sizes

	Method	1°	1/4°
(1)	Raw FITS input, not prefiltered[a]	315.0	194.0
(2)	Raw FITS input, prefiltered	42.0	25.9
(3)	Unstructured sequence-file input	9.2	4.2
(4)	Structured sequence-file input, prefiltered	4.0	2.7
(5)	SQL → unstructured sequence file input	7.8	3.5
(6)	SQL → structured sequence file input	4.1	2.2

[a]This method was not explicitly tested. The times shown were estimated by assuming linearity (see § 4.1.1. for justification) and scaling the second method's running time by the prefilter's reduction factor (7.5): *e.g.*, 42.0×7.5 and 25.9×7.5.

the number of files to be found on HDFS and the required time, which is justified given that the number of RPCs is constant per file and the amount of time required per RPC is constant.

4.1.2. Optimizing Hadoop with Sequence Files

While Hadoop is designed to process very large data sets, it performs better on a data set of a given size if the total number of files is relatively small. In other words, given two data sets of identical data with one data set stored in a large number of small

files and the other stored in a small number of large files, Hadoop will generally perform better on the latter data set. While there are multiple contributing factors to this variation in behavior, one significant factor is the number of RPCs required in order to initialize the MapReduce job, as illustrated in the previous section. For each file stored on HDFS, the client machine must perform multiple RPCs to the HDFS name node to locate the file and notify the MapReduce initialization routines of its location. This stage of the job is performed in serial and suffers primarily from network latency. For example, in our experimental database we processed 100,000 files. Each file had to be independently located on HDFS prior to running the Map-Reduce job and this location was performed serially over set of 100,000 files. Consequently, this step alone could take nearly 5 hr on our setup. Even in the prefiltered method described previously, this step still took about 30 minutes (see Fig. 7).

The preceding is a well-known limitation of the Hadoop system (White 2009) and Hadoop includes an *application programming interface* (API) and a set of data structures specifically targeted at alleviating it. The relevant structure is a *sequence file*. A sequence file contains an arbitrary number of key/value pairs, where a key is used to indicate (and locate) a specific file and the value is the file in question. In this way, individual files can be retrieved from a sequence file in a

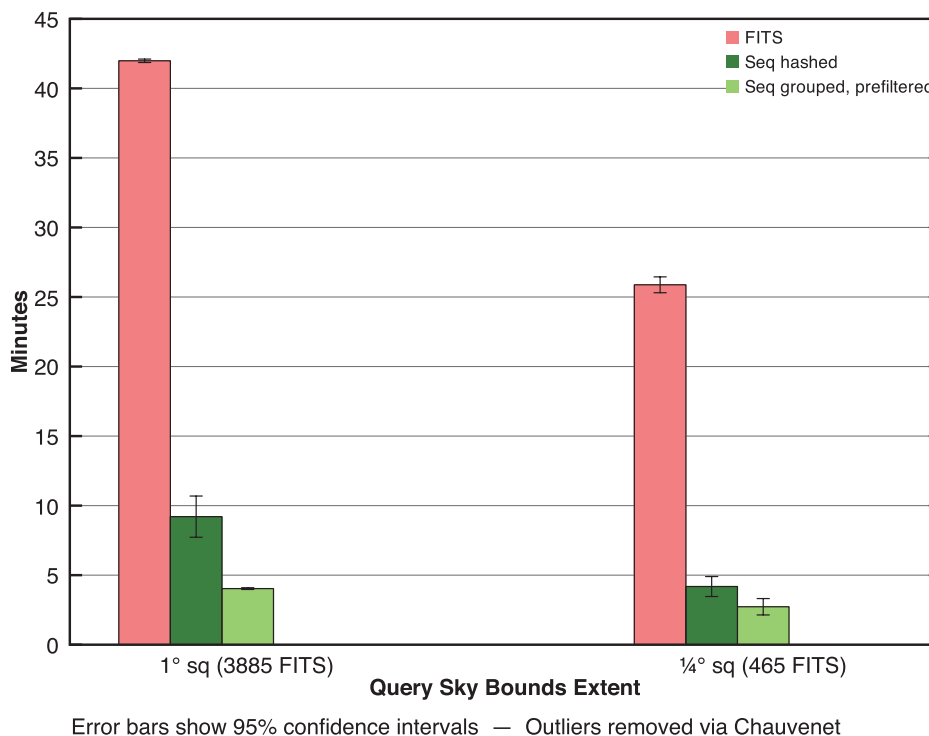Error bars show 95% confidence intervals — Outliers removed via Chauvenet

Fig. 6.—Co-add running times (shorter bars are better). Two queries (large on left, small on right) were run via three methods (from left to right: prefiltered FITS input, nonprefiltered unstructured sequence-file input, and prefiltered structured sequence-file input). We observe that the second and third methods yielded 5 times and 10 times speedups relative to the first method for the large query. Note that the number of mapper input records (FITS files) for the three methods for the large query was 13415, 100058, and 13,335, respectively.
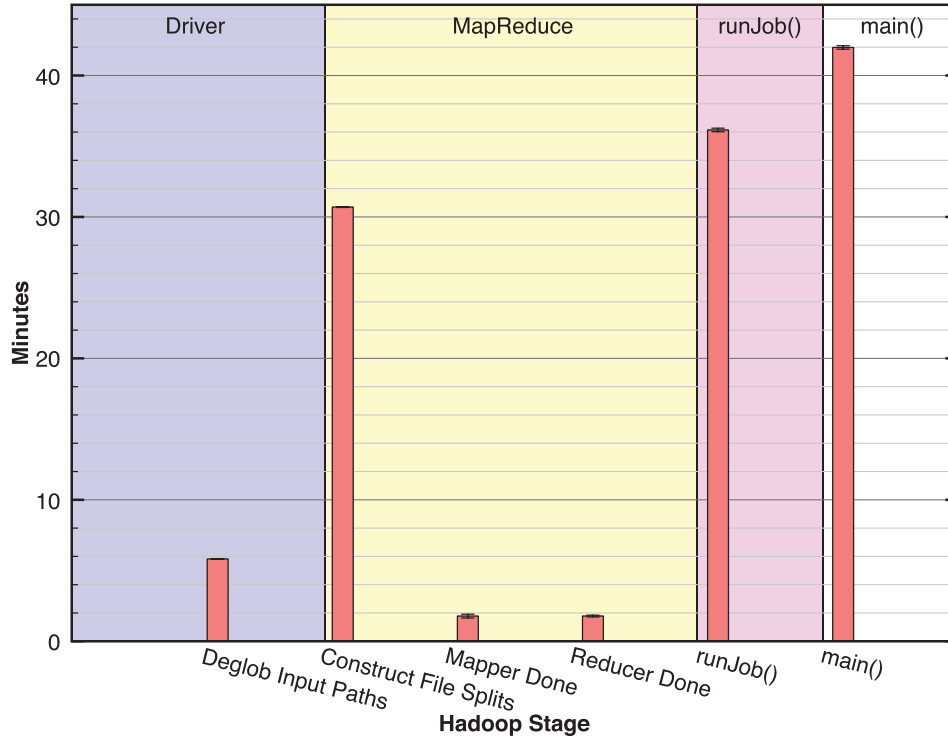
FIG. 7.—Breakdown of the overall running time for the FITS input and large query (see Fig. 6) into the salient stages. The last bar represents the total time, 42 minutes. A call to `main()` involves a preliminary setup stage in the *driver* followed by a single call `runJob()` that encapsulates the MapReduce process. Thus, the bar in the fourth region is the sum of the bars in the first and third regions. Likewise, the call to `runJob()` runs the *MapReduce* process and has been further broken down into the components shown in the second region. Thus, the bar in the third region is the sum of the bars in the second region. We observe that the dominating step of the overall process is *Construct File Splits*. This indicates that the Hadoop co-addition process spent most of its time locating the FITS files on HDFS and preparing them as input to the mappers. The actual image-processing computations are represented only in the two shortest bars labeled *Mapper Done* and *Reducer Done* indicating that Hadoop spent a small proportion of its total time on the the fundamental task. This observation suggests that there is substantial inefficiency in the associated method.

random-access fashion. In effect, a sequence file is a concatenation of smaller files bundled with an index.

We use sequence files by grouping many individual FITS files into a single sequence file, indexed simply by the FITS filename itself. Thus, we can retrieve a specific FITS file rapidly while simultaneously reducing the number of actual files on disk from the number of FITS files to the number of resulting sequence files. As a first strategy, we randomly grouped FITS files into sequence files (thus producing *unstructured* sequence files), as illustrated in Figure 8 (*top*). Table 4 and Figure 6 show the results of using this approach in row 3 of the table and bar 2 of each plot group. As the results show, this approach improved upon the prefiltered method (described previously) by a factor of 5 and theoretically improved upon the most naive approach by a factor of approximately 38. It should be noted that we never implemented the most naive method and so have no running times for it. Consequently, the prefiltered method as applied directly to the FITS database should be taken as the baseline for performance comparisons.

Note that unstructured sequence files outperform prefiltering of the direct FITS file data set by a factor of 5. Table 4 and

Figure 6 illustrate this fact. What is interesting is that this five-fold improvement occurs *despite* the fact that in the prefiltered case we actually reduced the input of 100,000 FITS files down to about 13,000 files, while in the sequence-file case we could not perform any prefiltering. In the unstructured sequence-file case, many more FITS files were processed relative to the pre-filtered FITS file case. Despite this inefficiency on part of the sequence-file method, it still significantly outperforms the pre-filtering method, due to its vast reduction in the number of actual files on disk: *i.e.*, the conversion of the input representation from a set of FITS files to a set of sequence files.

4.1.3. Unstructured and Structured Sequence Files

FITS files can be assigned to sequence files in any way the programmer chooses. In the previous section, we considered the worst-case performance by using an unstructured sequence-file database. The resulting sequence-file database could not be pruned in any meaningful fashion at runtime, *e.g.*, the prefiltering mechanism described in § 4.1.1., and therefore had to be fully read and processed during a MapReduce job. Results of
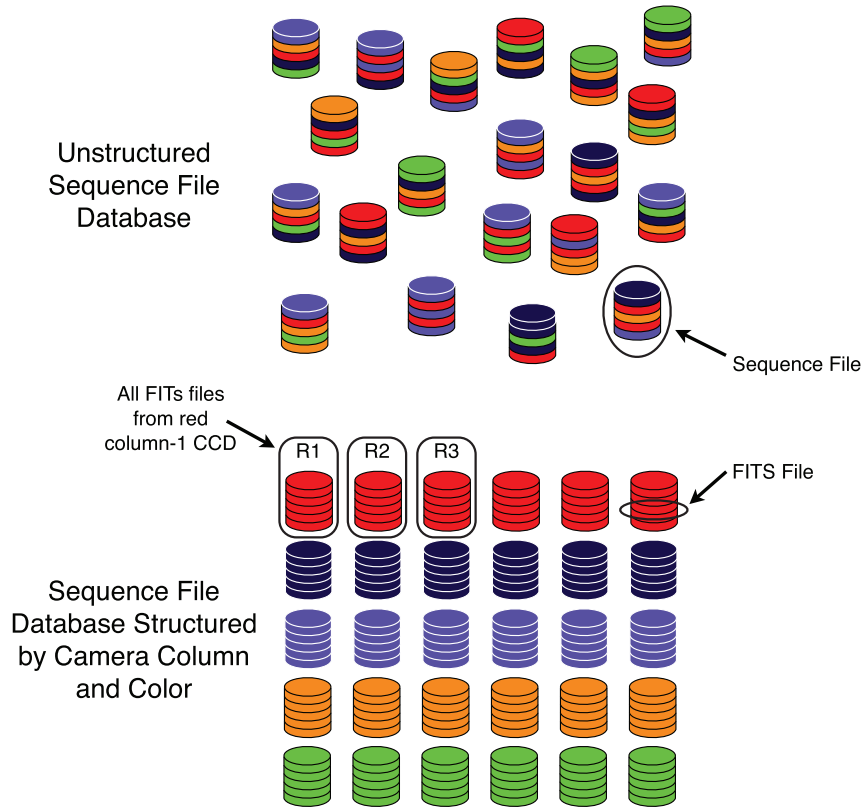
FIG. 8.—A sequence file (each column) comprises a set of FITS files (layers). We convert a database of numerous FITS fits to a database of fewer actual files (sequence files) using one of two methods. The first method has no structure (*top*). FITS files are assigned to sequence files at random. The second method maps the SDSS camera's CCD layout onto the sequence-file database; *i.e.*, there is one sequence file for each CCD on the camera. FITS files are assigned accordingly, and thus a given sequence file contains FITS files of only one bandpass and from only one column of the camera.

using such a sequence-file database were presented in the previous section.

However, it is also worth considering a more rationally motivated sequence-file structure, since doing so might permit pruning: *i.e.*, filtering as demonstrated previously. We once again consider the SDSS camera's CCD layout. This time, we use the SDSS camera's CCD arrangement not to define a prefiltering method on FITS files, but rather to impose a corresponding structure on the sequence-file database. We define 30 distinct sequence-file types, one corresponding to each CCD of the camera (see Fig. 8). Thus, a given sequence file contains only FITS files originating from one of the original 30 CCDs. If the sequence files are named in a way that reflects the glob filter described previously, we can then filter entire sequence files in the same way that we previously prefiltered FITS files. This method of filtering permits us to eliminate entire sequence files from consideration prior to MapReduce on the basis of bandpass or column coverage of the query. We anticipated a corresponding improvement in performance resulting from the reduction of wasted effort spent considering irrelevant FITS files in the mappers.

Table 4 (*row 4*) and Figure 6 (*third bar in each set*) show the results of using structured sequence files and prefiltering in concert. We observe a further reduction in running time over unstructured sequence files by a factor of 2 on the larger query. The savings in computation relative to unstructured sequence files resulted from the elimination of much wasted effort in the mapper stage spent considering and discarding those FITS files whose bandpass did not match the query or whose sky bounds did not overlap the query bounds.

Both prefiltering methods suffer from false positives, resulting from the fact that the spatial filter occurs on only one spatial axis. Likewise, the nonprefiltered method performs no prefiltering at all. Therefore, in all three methods, effort is still being wasted considering irrelevant FITS files. The prefiltering methods processed 13,000 FITS files in the mappers, while the nonprefiltered method processed the entire data set of 100,000 FITS files. However, the number of FITS files that actually contributed to the mosaic was only 3885. In the next section we describe how we used a *Structured Query Language* (SQL) database and query prior to running MapReduce to eliminate this problem.
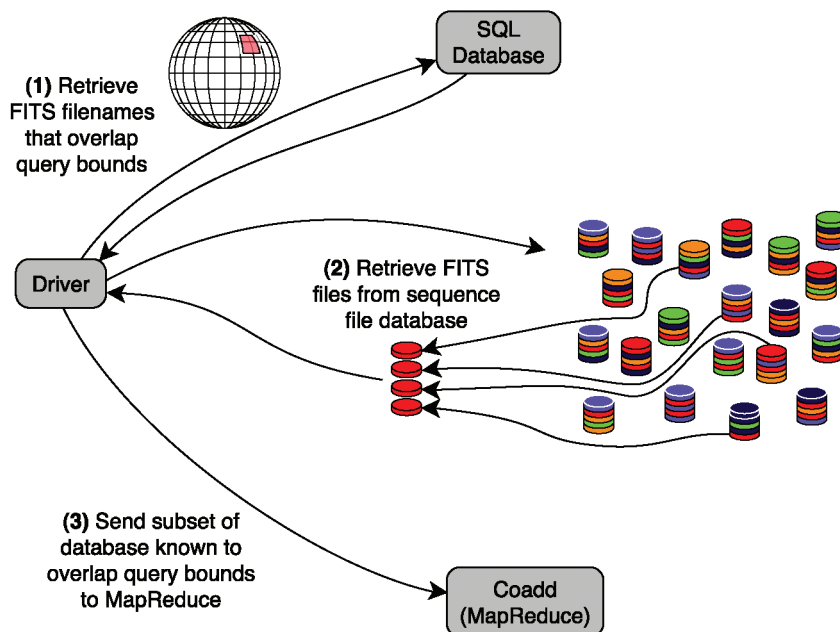
4.1.4. Using a Relational Database to Prefilter the FITS Files

All three methods described previously (prefiltered FITS files, unstructured sequence files, and prefiltered structured sequence files) process irrelevant FITS files in the mappers. Toward the goal of alleviating this inefficiency, we devised a new method of prefiltering. This method consists of building a relational database of the FITS files outside of HDFS and outside of Hadoop in general. The actual image data are not stored in the relational database—only the bandpass filter and sky bounds of each FITS file are stored, along with the necessary HDFS file reference data necessary to locate the FITS file within the sequence-file database: *i.e.*, its assigned sequence file and its offset within the sequence file. Running a job using this method consists of first performing a SQL query to retrieve the subset of FITS files that are relevant to the user's query and, from the SQL result, constructing a set of HDFS *file splits*[11] to specify the associated FITS files within the sequence-file database (see Fig. 9). The full set of file splits then comprises the input to MapReduce.

The consequence of using SQL to determine the relevant FITS files and sending only those FITS files to MapReduce is that this method does not suffer from false positives as de-

scribed previously. Thus, the mappers waste no time considering (and discarding) irrelevant FITS files, since every FITS file received by a mapper is guaranteed to contribute to the mosaic. The intention is clearly to reduce the mapper running time as a result. Table 5 shows the number of FITS files read as input to MapReduce for each of the six experimental methods. Note that prefiltering is imperfect; *i.e.*, it suffers from false positives and accepts FITS files, which are ultimately irrelevant to the coaddition task. However, the SQL methods only process the relevant files.

Figure 10 shows the results of the new method. We have removed the prefiltered FITS method from the plot (shown in Fig. 6) so that we may concentrate our attention on the faster methods: namely, the two sequence-file methods previously described (the first two bars shown in Fig. 10) and two new methods, one each for performing SQL against the unstructured sequence-file database (*bar 3*) and for performing SQL against the structured sequence-file database (*bar 4*).

Several patterns are observed in the plot. First, note that it is only meaningful to compare SQL vs. non-SQL for a given sequence-file database, either structured or unstructured. Thus, the appropriate comparisons to make are between bars 1 and 3 or between bars 2 and 4. We observe that in such comparisons, the SQL method does successfully outperform the non-SQL method in most cases, but to a lower degree than we hoped for when we implemented it; *i.e.*, bar 3 shows only a small improvement over bar 1, and bar 4 shows virtually no improvement over bar 2.

---

[11] Although the concept of a *file split* is complicated when one considers the full ramifications of storing files on a distributed file system, it suffices for our discussion to define a file split simply as the necessary metadata to locate and retrieve a FITS file from within a host sequence file.

TABLE 5
NUMBER OF FITS FILES PROCESSED BY MAPREDUCE

Method	1° (3885[a])	1/4° (465[a])
Raw FITS input, not prefiltered[b]	100058	100058
Raw FITS input, prefiltered	13415	6714
Unstructured sequence-file input	100058	100058
Structured sequence-file input, prefiltered	13335	6674
SQL →unstructured sequence-file input	3885	465
SQL →structured sequence-file input	3885	465

[a] Indicates coverage for each query (the number of FITS files from the data set that overlap the query bounds).

[b] This method was not explicitly tested, but since it performs no prefiltering, it should be clear that the entire data set would be processed.

Let us consider the results indicated by bars 2 and 4 for the larger query. The noteworthy observation is that while SQL shows some benefit, that benefit is lower than originally anticipated, given the 3.5 times difference in mapper input records (FITS files), 13,335 vs. 3885. 13,335 is the number of FITS files read in by the prefiltered sequence-file method (of which 3885 were relevant and the rest discarded), and 3885 is the number of FITS files read in by the SQL method (all of which were relevant). Therefore, we can conclude that in this case the cost of considering and discarding numerous irrelevant FITS files was negligible and likewise that the additional complexity imposed by supporting and using an external SQL database offers no benefit.

Let us consider the results indicated by bars 1 and 3 for the larger query. Again, we note little improvement in performance given the variance of the confidence intervals. This is surprising given the dramatic difference in mapper input records: a full 26-fold difference. The first bar represents nonprefiltered unstructured sequence files (thus, 100,058 input records), and the third bar represents a SQL method (thus, 3885 input records). The conclusion is similar in this case that the cost of discarding irrelevant files is low.

We might then ask: why is there a twofold difference in performance between the two SQL methods, considering that they both processed exactly the same amount of data? To answer this question we must investigate not merely the number of mapper input records, but the number of *mapper objects* launched by Hadoop. These two values are rarely equal, because a mapper can be reused to process multiple input records. In the SQL
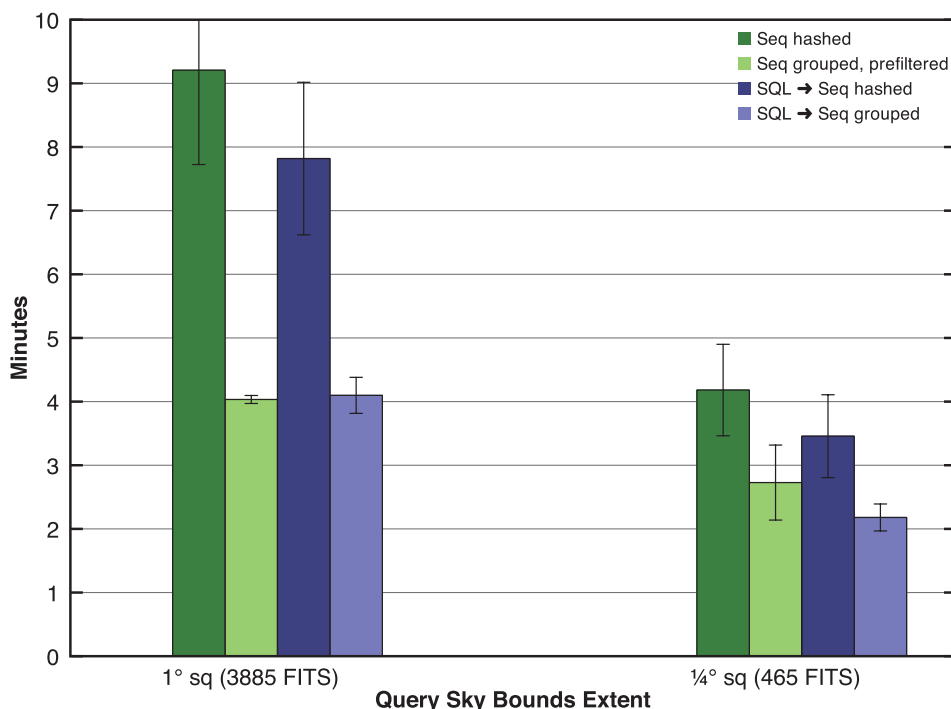

FIG. 10.—Co-add running times (shorter bars are better). Two queries (large on left, small on right) were run via four methods (from left to right: nonprefiltered unstructured sequence-file input, prefiltered structured sequence-file input, SQL unstructured sequence-file input, and SQL structured sequence-file input). The first two bars in each set correspond to the last two bars from Fig. 6, and the second two bars provide comparative running times using the SQL method on each of the two sequence-file types. For any given sequence-file type (bars 1 and 3 or bars 2 and 4), we observe little improvement resulting from the SQL method, despite the inefficiency suffered by the non-SQL method (consideration and discarding of irrelevant FITS files in the mappers). Furthermore, we observe that SQL performs quite differently on the two sequence-file types (compare bars 3 and 4), despite the fact that the two methods process an identical set of FITS files and perform an identical set of computations. Please see the text for an explanation of this variation in performance.

unstructured-sequence-file method, the 3885 input FITS files were processed by 1714 mapper objects (about two FITS files each), and in the SQL structured-sequence-file method, 338 mapper objects were used (about 11 FITS files each). This discrepancy is due to the way we assign FITS files to each mapper object. For each mapper object, we assign FITS files in the same HDFS block. Due to the replication of blocks across HDFS, copies of each block will be stored on multiple hosts. When possible, Hadoop will schedule mapper objects to run on one of the hosts where that mapper's input blocks are located so that the files can be efficiently accessed from local disks. Clearly, this approach generates a variable number of mapper objects, depending on the structure of the input sequence file. In the case of unstructured sequence files, the FITS files relevant to a given query are scattered throughout the sequence-file database, whereas in the case of structured sequence files, the relevant FITS files are more tightly packed. Thus, greater data locality is achieved in the assignment of FITS files to mappers in the structured case and fewer mapper objects are required. To complete our understanding of the difference in running times we must further consider the maximum number of mapper objects that the cluster can sustain simultaneously. Since a mapper represents a conceptual unit of computation, it corresponds to—or directly relies upon—a processor core in order to operate. Therefore, the number of simultaneously sustainable mapper objects for a Hadoop job is limited by the number of cores available on the entire cluster.[12] In our case, this value is about 800. We can now see one fundamental problem with the unstructured case: not all of the 1714 mappers could run simultaneously; some could not begin processing until after others had completed their own processing. This limitation was not true in the case of structured sequence files, where only 338 slots were required and they could all run simultaneously. However, one might actually predict superior performance from the unstructured case for the simple fact that it benefits from 800 times parallelism, whereas the structured case only benefits from 338 times parallelism. We theorize that the explanation for why such behavior was not observed lies in the nonnegligible startup cost of launching a mapper object; *i.e.*, there is a genuine benefit in reusing mappers, and there must be some tipping point in this tradeoff, where the benefit of additional parallelism is outweighed by the cost of creating mappers for brief computational needs.

5. CONCLUSIONS

This work presented our implementation and evaluation of image co-addition within the MapReduce data-processing framework using Hadoop. We investigated five possible meth-

---

[12] Ignoring, for the moment, additional factors such as distribution of core assignments between mapper and reducer objects and shared usage of the cluster by concurrent MapReduce jobs unrelated to our application.

ods of implementation, with the latter methods designed to improve upon the earlier methods.

Our first round of experiments processed a data set containing 100,000 individual FITS files. Despite the use of a prefiltering mechanism to decrease the input size, this method yielded poor performance due to the job initialization costs. To decrease the initialization time we must process a data set consisting of fewer actual files, but obviously without altering the underlying data. This goal is achieved through the use of sequence files that group individual files together. Sequence files are offered through the Hadoop API for precisely this purpose.

Our next round of experiments considered sequence files that were grouped in an unstructured manner and that therefore could not be prefiltered. Despite this weakness relative to the first method, the use of unstructured sequence files still yielded a fivefold improvement in performance.

We then ran experiments on sequence files that were grouped in ways that reflected the same prefiltering mechanism used earlier. The sequence files themselves could then be prefiltered. This method yielded another twofold improvement in performance (10-fold over the original method).

The first three methods described all suffered from an inefficiency whereby irrelevant FITS files were considered and discarded in the mapper stage. Toward the goal of alleviating this inefficiency we devised our fourth and fifth methods, which used a SQL query prior to MapReduce to identify the relevant FITS files. Those FITS files were then retrieved from the two previously described sequence-file databases to run MapReduce. When applied to the unstructured sequence-file database, this SQL method yielded minimal improvement in performance. The lack of structure of the sequence files prevented the efficient reuse of mapper objects. However, when the SQL method was used in conjunction with the structured sequence-file database, the mappers were reused more efficiently and good performance was achieved, although it should be noted that the SQL method still did not significantly outperform the non-SQL method, in which bandpass filter and one-dimensional spatial prefiltering were used. In the case of the smaller query there was some advantage, however.

It should be noted that while SQL showed little improvement over prefiltering in our current experiments, we anticipate that SQL should show significant advantage on larger data sets. In such a case, prefiltering would be more adversely affected by the false-positive problem; *i.e.*, more irrelevant FITS files would pass the prefilter to be considered and discarded by the mappers. There ought to be a data set size above which this wasted effort begins to show measurable degradation in performance. On the other hand, the SQL method should show comparatively more gradual decrease in performance as the data set grows, since it only processes the relevant FITS files to begin with. This prediction is admittedly speculative as we have not yet conducted the necessary experiment.

The use of Hadoop shows great promise for processing massive astronomical data sets. On a 400 node cluster we were able to process 100,000 files (300 million pixels) in 3 minutes using two different methods: one with SQL and one without.

6. FUTURE WORK

Our current work represents merely the early stages of our research into the use of Hadoop for image co-addition. Areas of future work include:

1. Conversion of the mapper and reducer code to C++ and incorporation of our existing C++ library of image-processing routines that are capable of performing much more sophisticated co-addition algorthms.

2. The addition of time bounds to the query parameters so that co-adds may be generated only within a specified window of time. This behavior will enable the investigation of time-variable phenomena such as variable stars and moving objects: *e.g.*, asteroids.

3. The addition of subsequent pipeline stages to process the resulting co-adds, such as the detection and tracking of moving and variable objects.

4. The development of new parallel machine-learning algorithms for anomaly detection and classification.

Ultimately, we intend to develop our system into a full-fledged data-reduction pipeline for petabyte astronomical data sets, as will be produced by next-generation sky surveys such as LSST.

REFERENCES

Bertin, E., et al. 2002, in ASP Conf. Ser. 281, Astronomical Data Analysis Software and Systems XI, ed. D. A. Bohlender, D. Durand, & T. H. Handley (San Francisco: ASP), 228

Dean, J., & Ghemawat, S. 2004, in OSDI'04: Sixth Symposium on Operating System Design and Implementation (publisher's location?: publisher's name?), 137

Gunn, J. E., et al. 1998, AJ, 116, 3040

Jacob, J., et al. 2009, IJCSE, 4, 73

Szalay, A. S., Connolly, A. J., & Szokoly, G. P. 1999, AJ, 117, 68

White, T. 2009, Hadoop: The Definitive Guide (Sebastopol: O'Reilly Media), 103

Queries

1. The right running title was shortened to meet the limit of 50 characters/spaces. Please make different adjustments if desired.
2. Please add a reference with full publication information for the 2004 paper by Google introducing MapReduce or reword this sentence to avoid the "Google 2004" citation..
3. Several references did not meet requirements to be references and were changed to footnotes. Please adjust the wording as needed.
4. Both "1/4° square" and "1° square" must be adjusted to PASP style. Would $\frac{1}{4}\deg^2$ or $\frac{1}{4}° \times \frac{1}{4}°$ best convey your meaning?
5. Please clarify 4/6 strips and 3/6 strips. If they are measurements, please include the unit (inch, mm, cm, etc.). If it means "four of the six strips" and "three of the six strips," please adjust the text as applicable.
6. Please define HDFS at its first use.
7. Please replace the word "below" here.
8. Please adjust the wording to avoid beginning the sentence with a digit or having to spell out Thirteen thousand, three hundred and thirty five.
9. Please provide the publisher's name and location for Dean & Ghemawat. Also note that some tracked changes became untracked when the reference list was rearranged for alphabetical order.
10. All algorithms were manually reset to journal style and renumbered as tables. Please check them for accuracy and please check that all adjusted table citations are correct.
11. The bold headlines for each caption were removed, because most of them duplicated information that followed in the same caption. Please adjust any captions as needed.
12. Please add a (bottom) to Fig. 8 or remove the (top).
13. Because the steps are detailed in full in the figure, the duplicate detailed steps were removed from the caption. Please make different adjustments as needed.