The VLDB Journal (2012) 21:589–609
DOI 10.1007/s00778-012-0279-5

SPECIAL ISSUE PAPER

On the optimization of schedules for MapReduce workloads in the presence of shared scans

Joel Wolf · Andrey Balmin · Deepak Rajan · Kirsten Hildrum · Rohit Khandekar · Sujay Parekh · Kun-Lung Wu · Rares Vernica

Received: 15 August 2011 / Revised: 22 February 2012 / Accepted: 8 May 2012 / Published online: 10 June 2012
© Springer-Verlag 2012

Abstract We consider MapReduce clusters designed to support multiple concurrent jobs, concentrating on environments in which the number of distinct datasets is modest relative to the number of jobs. In such scenarios, many individual datasets are likely to be scanned concurrently by multiple Map phase jobs. As has been noticed previously, this scenario provides an opportunity for Map phase jobs to *cooperate*, sharing the scans of these datasets, and thus reducing the costs of such scans. Our paper has three main contributions over previous work. First, we present a novel and highly general method for sharing scans and thus amortizing their costs. This concept, which we call *cyclic piggybacking*, has a number of advantages over the more traditional *batching* scheme described in the literature. Second, we notice that the various subjobs generated in this manner can be assumed in an optimal schedule to respect a natural chain precedence ordering. Third, we describe a significant but natural generalization of the recently introduced FLEX scheduler for optimizing schedules within the context of this cyclic piggybacking paradigm, which can be tailored to a variety of cost metrics. Such cost metrics include average response time, average stretch, and any minimax-type metric—a total of 11 separate and standard metrics in all. Moreover, most of this carries over in the more general case of overlapping rather than identical datasets as well, employing what we will call *semi-shared* scans. In such scenarios, chain precedence is replaced by arbitrary precedence, but we can still handle 8 of the original 11 metrics. The overall approach, including both cyclic piggybacking and the FLEX scheduling generalization, is called CIRCUMFLEX. We describe some practical implementation strategies. And we evaluate the performance of CIRCUMFLEX via a variety of simulation and real benchmark experiments.

Keywords MapReduce · Shared scans · Scheduling · Allocation · Optimization · Amortization

Rares Vernica is partially supported by NSF grant 0910989.

J. Wolf (✉) · K. Hildrum · K.-L. Wu
IBM T.J. Watson Research, Hawthorne, NY 10532, USA
e-mail: jlwolf@us.ibm.com

K. Hildrum
e-mail: hildrum@us.ibm.com

K.-L. Wu
e-mail: klwu@us.ibm.com

A. Balmin
IBM Almaden Research, San Jose, CA 95120, USA
e-mail: abalmin@us.ibm.com

D. Rajan
Lawrence Livermore National Laboratory,
Livermore, CA 94550, USA
e-mail: rajan3@llnl.gov

R. Khandekar
Knight Capital Group, Jersey City, NJ 07310, USA
e-mail: rkhandekar@gmail.com

S. Parekh
Bank of America, New York, NY 10080, USA
e-mail: sujay.parekh@bankofamerica.com

R. Vernica
HP Laboratories, Palo Alto, CA 94304, USA
e-mail: rares.vernica@hp.com

1 Introduction

Google's MapReduce [8] and its open-source implementation Hadoop [10] have become highly popular in recent years. There are many reasons for this: simplicity, automatic parallelizability, natural scalability, and implementability on

commodity hardware. Important built-in features include fault tolerance, communications, and scheduling.

We focus on the problem of scheduling MapReduce work in this paper and more specifically on a specialized but common and important variant first introduced by [1]: optimizing the amortized costs of shared scans of Map jobs. Before describing this "shared scan" problem in detail, we provide a brief overview of some popular *generic* MapReduce schedulers. Understanding the original MapReduce scheduling problem and its history motivates our approach to the shared scan special case and places our solution to that problem in the proper context.

Early MapReduce implementations, including Hadoop, employed *First In First Out* (FIFO) scheduling. But while simple and almost universally applicable, FIFO is known to have problems with job starvation in most environments. A large job can "starve" a small job that arrives even modestly later. Worse, if the large job is a batch submission and the small job is an ad-hoc query, the exact completion time of the large job would not be particularly important, while the completion time of the small job would be.

The Hadoop *Fair Scheduler* (FAIR) is a slot-based MapReduce scheme designed to avoid starvation by ensuring that each job is allocated at least some minimum number of slots [28–30]. (Slots are the basic unit of resource in a MapReduce environment.) But FAIR does not attempt to actually optimize any specific scheduling metric. And a schedule designed to optimize one metric generally performs quite differently from a FAIR schedule or one designed to optimize another metric. By contrast, the *Flexible Scheduler* (FLEX) [26] can optimize a wide variety of standard scheduling theory metrics while ensuring the same minimum job slot guarantees as FAIR, and maximum job slot guarantees as well. The desired metric can be chosen from a menu that includes response time, stretch, and any of several metrics that reward or penalize job completion times compared to possible deadlines. This last includes the number of tardy jobs, tardiness, lateness, and also *Service Level Agreements* (SLAs). There are 16 combinatorial choices in all, because the metrics can be either weighted or unweighted, and one can optimize either their average (or, equivalently, from the perspective of optimization, their sum) across all jobs, or the maximum such value. Moreover, FLEX can be regarded as an add-on module that sits on top of FAIR, works synergistically with it, and employs its extensive infrastructure. FLEX performs very well on all metrics relative to both FAIR and FIFO [26].

As first pointed out in [1], there is another, more subtle opportunity for improved scheduling in many common MapReduce environments: optimizing the amortized costs of shared scans of jobs in their Map phase. (We will use the term *Map jobs* for jobs in their Map phase from here on.) This is because the number of *distinct* datasets is often modest relative to the number of MapReduce jobs. A particular MapReduce dataset may be used simultaneously by multiple Map jobs. Also, for many MapReduce jobs, the execution time cost of the Map phase is primarily that of scanning the data. Furthermore, the Map phase is often the most expensive (and sometimes the *only*) phase of a MapReduce job. Given all of the above, there is considerable leverage in having Map jobs *cooperate* in some fashion on dataset scans, thus amortizing the costs of these scans.

In fact, [1] introduced a pair of schedulers designed for MapReduce environments with shared scans. (The schedulers in question will henceforth be known collectively as AKO, for the authors Agrawal, Kifer, and Olston.) The AKO schemes determine, for each popular dataset, an optimized *batching interval*. The idea is that Map jobs associated with this dataset delay starting work until this interval expires, and the scans of all the delayed Map jobs are then *batched* together, so that a single scan can be performed for all of them. An AKO off-line optimization algorithm assumes Poisson arrivals of known rates for the jobs associated with each dataset and uses a heuristic scheme based on Lagrange multipliers to find batching intervals that minimize either the average or the maximum value of a metric somewhat analogous to stretch. There are some limitations to the AKO approach.

1. Batching forces the trade-off of efficiency for latency. In other words, batching a number of scans together causes them to be delayed. A larger batching interval is more efficient but causes a longer average delay.
2. The assumption of Poisson arrivals allows the optimization to be performed, but is restrictive. Jobs do not always arrive according to such a distribution.
3. The assumption that the arrival rates of the jobs can be known in advance is problematic. These estimates are likely to be fairly rough approximations and thus may affect the quality of the optimization solution.
4. The schedule produced is inherently static and thus cannot react dynamically to changing conditions. (In fairness, the AKO implementation is more dynamic than the decisions produced by the scheduler itself.)
5. The AKO scheme optimizes two variants of an unusual scheduling metric known as *perceived wait time (PWT)*. This is defined as the difference between actual response time of a job and its minimum possible response time. (By contrast, optimizing the more natural metric *stretch*, to which the authors briefly allude, is the ratio of the two terms. But it is noted that optimizing stretch was deemed too difficult.) So, AKO optimizes either average or maximum PWT.
6. While average and maximum PWT are metrics that try to philosophically capture the spirit of fairness, the AKO scheme does not specifically deal with minimum slot

allocation constraints. These minimum constraints are a key guarantee in both FAIR and FLEX.

Our goal in this paper is to provide an amortizing MapReduce scheduler without these limitations. We eliminate latency due to batching by employing a different shared scan concept which we call *cyclic piggybacking*. Since cyclic piggybacking treats the datasets circularly rather than linearly, the advantages of amortization are achieved without the disadvantages of latency: Map jobs can begin immediately. There is no need for Poisson assumptions or accurate job arrival rate data. The scheme is dynamic rather than static, simply dealing with Map jobs as they arrive. Indeed, cyclic piggybacking itself does not involve any optimization at all. It can be performed entirely on the fly. Finding high-quality slot allocations among the jobs does still require an optimization scheme, and our new scheme for this is a generalization of the FLEX algorithm that decomposes each Map job into multiple *subjobs* based on the cyclic piggybacking. We then notice a natural chain precedence ordering that can be assumed among these subjobs in the optimal solution and solve a scheduling problem with these constraints. A total of 11 of the 16 original FLEX scheduling metrics can be optimized in this manner. (There may be heuristics available for the others.) The 11 include the important and commonly employed *minisum* metrics of average (or total) response time, stretch, and all of the *minimax* metrics. (Maximum stretch and makespan are common examples.) All of these metrics can either be weighted or unweighted.

Furthermore, the shared scan scenario considered above can be generalized significantly while retaining the spirit of the original design. Consider *semi-shared* scans, the case where jobs scan arbitrarily overlapping datasets, perhaps within one or more directories. Such a scenario would occur quite naturally, for instance, if one job scans a day of data, another scans a week, and a third scans a month. (Note that weeks are not necessarily contained within a single month.) Considering the obvious Venn diagram, the point is that there is a natural partitioning of the union of the datasets based on the overlapping subsets of various cardinalities. Although the term becomes something of a misnomer in this generalized context, cyclic piggybacking can be easily extended to the case of semi-shared scans. It also turns out that one can assume in an optimal schedule a natural precedence order among the overlapping subsets, from more overlapped to less overlapped. This is not a *chain* precedence scenario, as it is for shared scans. But, at least for the case of minimax scheduling metrics, the CIRCUMFLEX scheduling scheme works as is. So, one can optimize any of the 8 separate objective functions in this general scenario, including makespan and maximum stretch.

Notions similar to batching and cyclic piggybacking have been proposed in various contexts. For example, CoScan [25], a MapReduce scheduling framework, merges Pig jobs working on the same datasets in order to reduce I/O costs and eliminate redundant data processing. It also attempts to maximize the rewards for meeting user specified soft-deadlines for the jobs. The merging technique employed essentially batches jobs, so it suffers from the same latency problems that batching does at the MapReduce level. Other examples include the broadcast delivery of digital products [27] and databases [6,11,20,31]. To our knowledge, these have all been for the non-overlapping case: There does not appear to have been any work on the general semi-shared scenario. There seems to be very limited additional work in the area of sharing for MapReduce jobs. Note, however, that [17] examines a variety of sharing alternatives, including shared scans, in a MapReduce framework. In fact, their MRShare scheme also merges jobs into batches and evaluates these batches as single queries.

We call our shared scan scheduling scheme CIRCUMFLEX. For ease of exposition, we focus on the non-overlapping dataset scenario. But each component of CIRCUMFLEX can be generalized to the overlapping case, and we will outline these extensions as well.

In summary, the CIRCUMFLEX scheme described in this paper involves three main contributions.

1. We adapt the cyclic piggybacking method to MapReduce jobs for amortizing the cost of the shared scans and generalize it to handle *overlapping* datasets. This approach has a number of advantages over the batching scheme described in [1].
2. We notice that in the shared scan case, the various subjobs generated in this manner can be assumed to have a natural chain precedence order in the optimal schedule. In the semi-shared scan case, a natural precedence order can be assumed, though not chain precedence.
3. This precedence ordering allows us to formulate and solve a scheduling problem that is a generalization of FLEX. We optimize the scheduling of the subjobs with respect to any of a choice of 11 standard metrics in the shared scan case and 8 in the semi-shared scan case, while respecting minimum slot and precedence constraints.

We illustrate the excellent performance of CIRCUMFLEX relative to FLEX and BATCH in simulation experiments. Since the metrics optimized by AKO and CIRCUMFLEX are disjoint, we choose not to compare them directly. That would seem unfair to one scheme or the other. It would also be unfair to compare CIRCUMFLEX with FAIR or FIFO, because the latter two do not attempt to optimize any particular metric. We compare instead the quality of CIRCUMFLEX with that of FLEX, and with a batching scheme of our own devising. (This scheme, called BATCH, actually employs a FLEX optimization algorithm on the batched datasets.) Thus, we are

examining the performance benefits of high-quality cyclic piggybacking schedules to schedules optimizing the same metric, both with and without batching.

We also implemented CIRCUMFLEX inside Clydesdale [3,14], a very efficient, very I/O-bound research prototype for structured data processing on Hadoop. Clydesdale's efficiency means that the bulk of the processing time is spent scanning the data. This makes it a perfect application for shared scans, either batching or cyclic piggybacking. For example, CIRCUMFLEX on Clydesdale achieves up to 3 times better makespan and up to 5 times better average response time as compared to the sequential execution of the same workload. Our performance evaluation on Clydesdale also shows that CIRCUMFLEX significantly outperforms all batching schemes.

The remainder of this paper is organized as follows. In Sect. 2, we briefly describe cyclic piggybacking as an alternative employed by CIRCUMFLEX in favor of batching. Section 3 gives some preliminaries involving MapReduce and theoretical scheduling, essentially background information to understand the next section. We then present an overview of the CIRCUMFLEX scheduling algorithm in Sect. 4. Section 5 describes a CIRCUMFLEX implementation of shared scans in Hadoop. In Sect. 6, we describe both simulation and real benchmark experiments. Conclusions are in Sect. 7.

2 Batching and CIRCUMFLEX cyclic piggybacking

Perhaps, the most instructive way to describe both batching and cyclic piggybacking is via a common example. We do this before giving a formal definition of each approach. Specifically, we employ a simple example with two datasets.

See Figs. 1 and 2, which illustrate both schemes. The horizontal line in the center of each figures represents time. Two Map jobs scan one or the other of the datasets, labeled as dataset 1 and 2, respectively. Arrivals of jobs for the dataset

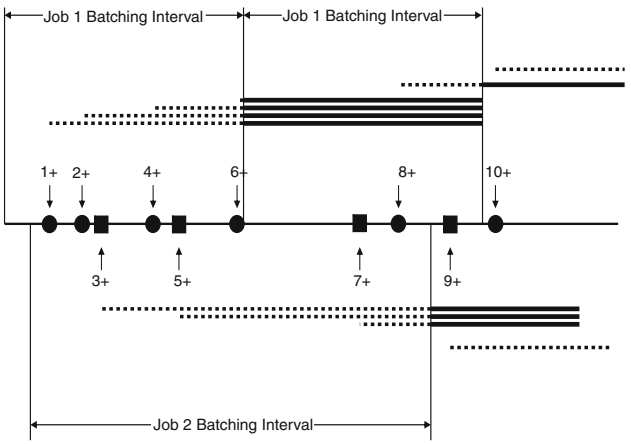

Fig. 1 Batching

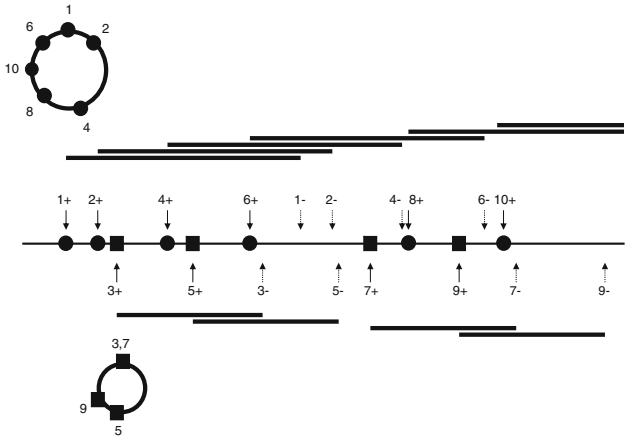

Fig. 2 Cyclic piggybacking

1 are depicted with circles in the center time lines. Details for dataset 1 are illustrated in the top halves of each figure. Similarly, arrivals of jobs for dataset 2 are depicted with squares in the center time lines, and details are given in the bottom halves. The figures thus depict the arrival of 10 Map jobs, 6 of which scan dataset 1 and 4 of which scan the dataset 2. The jobs are numbered, with a plus sign indicating their arrivals. Recall that we are considering first a non-overlapping dataset scenario. However, at least our cyclic piggybacking ideas can easily be extended to the general case, with *semi-shared scans*.

2.1 Batching

Consider Fig. 1. Optimized batching intervals for both datasets are computed via a scheme such as [1]. The vertical lines in the top half of the figure depict the temporal boundaries of the batching intervals for dataset 1, while the vertical lines in the bottom half depict the boundaries of a batching interval for dataset 2. The batching intervals for dataset 1 are shorter than the batching intervals for dataset 2, and only one complete batching interval for the latter dataset fits in the figure. The dotted lines represent the latency of each job incurs, and the batching itself is indicated via solid lines. Jobs 1, 2, 4, and 6 for dataset 1 arrive before the end of the first batching interval, and they are scanned together, as a batch with concurrency level 4, at the beginning of the second interval. Note that job 1 incurs a latency of nearly the entire dataset 1 batching interval. Job 8 arrives during the second batching interval and is scanned alone at the beginning of the third interval. Again, the dotted line illustrates the latency and the solid line illustrates this trivial (concurrency level 1) batch. Job 10 arrives during the third batching interval, and its scan is not shown in the figure. Jobs 3, 5, and 7 for dataset 2 are scanned with concurrency level 3. Job 9, which arrives in a subsequent batching interval, is not scanned in the figure.

The trade-off between latency and efficiency is illustrated in Fig. 1. Longer batching intervals allow for more jobs to be batched together, but the average latency of the jobs increases. The expected average latency is half the time in a batching interval under typical arrival rate assumptions. This is the fundamental trade-off of batching. On one extreme, a batching interval of nearly zero time results in almost no batching at all. This extreme incurs very small latencies, but is inefficient because all concurrency levels are close to 1. On the other extreme, a very large batching interval results in big concurrency levels and hence great efficiency. But latency becomes similarly huge.

Formally, the batching schemes described in [1] compute, for each dataset d, an optimized batching interval time T_d. The optimization algorithm assumes Poisson arrivals of jobs, as well as estimates of the rates associated with each dataset. It uses a Lagrange multiplier-based heuristic to find batching intervals that minimize either the average or maximum PWT. Assuming a start time of 0, time is then partitioned for any dataset d into multiple intervals of the form $[kT_d, (k+1)T_d)$ for each non-negative integer k. Any Map job arriving in interval $[kT_d, (k+1)T_d)$ and involving dataset d is then performed using a batched scan starting at time $(k+1)T_d$.

2.2 Cyclic piggybacking

Our new notion of *cyclic piggybacking* is best understood by considering Fig. 2, employing the same example as for batching in Fig. 1. If one thinks of a dataset as an ordered list of blocks, the dataset can be viewed linearly. Though it is something of a simplification, one can think of blocks as being scanned in this line segment from left to right, from first block to last block. However, recognizing that there is no special meaning to the first or last blocks, one can also "glue" these two blocks together and view the dataset *cyclically*. A good analogy here would be a clock, with blocks corresponding to hours. So, blocks can be scanned in a clockwise manner, starting with the first block—and as the scan reaches the last block, it can simply begin again at the first block. In the figure, the topmost point of a circle indicates the boundary between the first and last block. In the clock analogy, this point is simply "midnight."

At time 1, the first job for dataset 1 arrives. This is illustrated both via the linear time line and in a cyclic view of dataset 1 shown at the top. (A plus sign in the linear view indicates a job arrival, as before, while a minus sign refers to a job departure. In the cyclic view, these occur at identical points on the circle.) Map job 1 starts to scan data in clockwise fashion from the midnight starting point denoted by 1. Subsequently, a second job for dataset 1 arrives at time 2; this arrival is shown in both the linear and cyclic views. Considering the cyclic view, the clockwise arc from point 1 to point 2 involves previously scanned blocks, but job 2 can now *piggyback* its data scan of subsequent common data onto the remaining scan of job 1, amortizing costs. In the linear view, one notices that the concurrency level increases to 2 once job 2 starts. When job 1 completes its scan, the remaining blocks of job 2 can be scanned. In the absence of subsequent arrivals, the concurrency level would return to 1 during this portion of the scan. But notice that all of the concurrency levels depend dynamically on both future arrivals and departures: An arrival increments the concurrency level by 1, while a departure decreases it by 1. The subsequent arrival of a third job, this time for dataset 2, causes the cyclic view of dataset 2 at the bottom and the aligned linear view of job 3 below the center time line. The arrival of job 4 for dataset 1 causes a concurrency level of 3 for that dataset. The arrival of job 5 for dataset 2 causes a concurrency level of 2 for the 2nd dataset. The arrival of job 6 causes a concurrency level of 4 for the 1st dataset. Then, the departure of job 3 occurs, reducing the concurrency level back to 1 for the 2nd dataset. Note that the eventual departure of job 5 for dataset 2 and the subsequent arrival of job 7 for dataset 2 causes a new single scan of the *first* blocks of the 2nd dataset again, and so forth.

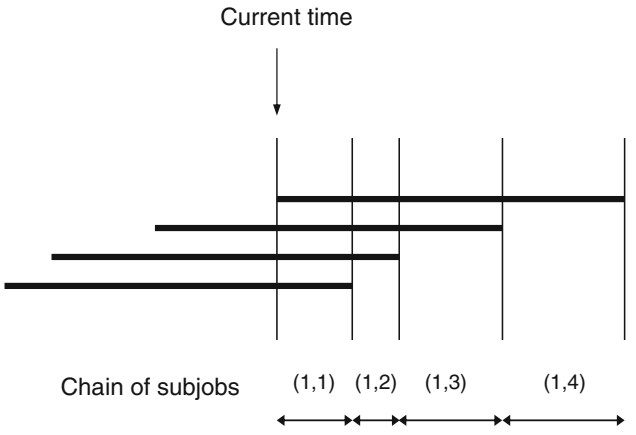

Fig. 3 Creation of chain precedence constraints for shared scans

See Fig. 3, which illustrates the state of affairs for dataset 1 at precisely the time when job 6 (the 4th job for dataset 1) arrives. At this moment in time, 3 jobs for dataset 1 are in the process of being scanned. The figure shows a decomposition of the current remaining work for the 1st dataset into 4 subjobs, labeled $(1, 1), \ldots (1, 4)$. Subjob $(1, 1)$ starts at the current time and completes at the end of the scan of the 1st job for dataset 1. The concurrency level is 4. Subjob $(1, 2)$ starts when subjob $(1, 1)$ ends and completes at the end of the scan of the 2nd job for dataset 1. The concurrency level is 3.

Formally, subjob (d, k) for dataset d starts when subjob $(d, k-1)$ ends and completes at the end of the scan of the kth job for dataset d. It has concurrency level $K_d - k + 1$, where K_d is the total number of currently active jobs for dataset d. (In this case of Fig. 3, $K_1 = 4$.)

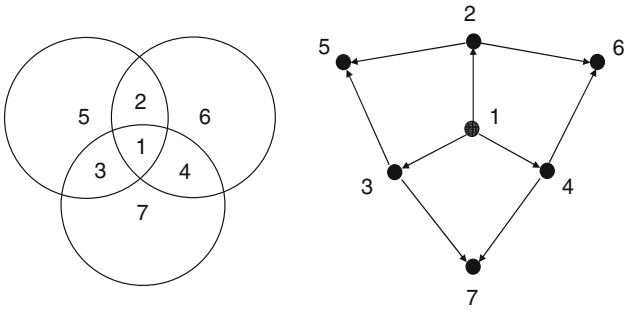

Fig. 4 Creation of precedence constraints for semi-shared scans

A few rather delicate points should be made here. First, note that each subjob belongs to both a Map job and a dataset. The notation clearly references the latter rather than the former. Because calling (d, k) a subjob of job d overuses the term "job," we avoid this. In other words, we reserve the word job for the Map jobs themselves. Note also that the notation is quite dynamic—it changes with job arrivals. Finally, there could also many *synonyms* for subjobs, depending on the job under consideration: If jobs j_1 and j_2 represent two successive scans for the same dataset d, then subjob (d, k) viewed via the first job is the same as subjob $(d, k-1)$ viewed via the second. To avoid this unsatisfactory notation, we insist on using the subjob terminology associated with the last job to arrive for a particular dataset. (See Fig. 3.) We will discover in Sect. 4, via a simple interchange argument, that the optimal way to complete these subjobs is in sequential order. In the figure, this means that subjob $(d, 1)$ should be completed before subjob $(d, 2)$ starts, and so on.

In the case of semi-shared scans, the subjobs are modestly more complex. Consider the left-hand side of Fig. 4, representing the remaining scans required for three separate but overlapping arrivals at the time of the third arrival. The 6 natural subjobs created in this example correspond to the subsets in the figure. It will turn out in Sect. 4 that the optimal way to complete these subjobs is from the most overlapped to the least overlapped subset. We have simply numbered these 6 subjobs in topological order and displayed the precedence graph shown in the right-hand side of the figure. So, the most overlapped subjob 1 should precede subjobs 2, 3, and 4 (which are not comparable). Similarly, subjobs 2 and 3 should precede subjob 5, and so on. (Subjobs 5, 6, and 7 are also not comparable.)

Defined in this manner, cyclic piggybacking suffers from none of the first four disadvantages noted for AKO in Sect. 1. There is no built-in latency, since jobs are ready to start their Map phase scans instantly. The arrival distribution and rates are irrelevant. The design is completely dynamic, reacting on the fly to any new job arrivals: No optimization algorithm needs to be executed.

Of course, nothing is ever quite as simple as it might appear at first. The description of cyclic piggybacking above is a bit too simple in several ways. We introduce these issues now, resolving them in subsequent sections.

First, by describing the scanning of datasets in either linear or cyclic terms, we have essentially implied a discrete block ordering that does not really exist. This simplification is strictly for purposes of exposition. In fact, no real order for the blocks exists, except that inherently implied by the optimality of the subjob sequencing. There is great flexibility built into the MapReduce paradigm: The actual block scan execution order within a subjob depends on an entirely different layer of the MapReduce scheduler. (This so-called *assignment* layer, described in Sect. 3, considers issues such as data locality when assigning a Map scan to an available slot on a particular node.) In any reasonable implementation of the cyclic piggybacking scheme, there will simply be a bit for each active job and relevant block, which notes whether or not that block has already been scanned for that job. Within a subjob, the scanning order of the blocks is essentially immaterial.

Although we say that the blocks are *ready* immediately for scheduling, we may theoretically not have the resources to schedule that work. We have not yet introduced our MapReduce scheduler, which attempts to optimally allocate the slot resources to the various jobs and subjobs. (This scheduler will, indeed, attempt to start jobs as they arrive, based on minimum slot constraints.) But the bottom line is that our description of cyclic piggybacking thus far blurs this detail.

While we are effectively eliminating initial job latency via cyclic piggybacking, it is less clear how this new scheme compares with batching with respect to efficiency. Recall that batching can span the spectrum with respect to efficiency, depending on how large the batching intervals are. But it should be clear that a batching interval of 0 length, yielding no batching at all, is far less efficient than cyclic piggybacking while similarly avoiding latency. We will have more to say about these efficiency levels in Sect. 6.

Having described cyclic piggybacking itself, we now turn to the scheduling aspects of CIRCUMFLEX. We introduce this via scheduling preliminaries in Sect. 3. With this as background, we then describe the details of the CIRCUMFLEX scheduler in Sect. 4.

3 Scheduling preliminaries

In this section, we will give brief overviews of a number of MapReduce and scheduling theory concepts. Understanding these will simplify our exposition of the CIRCUMFLEX scheduler in Sect. 4. We will outline the two MapReduce scheduling layers, the scheduling metrics we consider and their usefulness, the notions of *parallelizable* jobs and

speedup functions, the theoretical notions of *moldable* and *malleable* scheduling, and finally, the concept of *epoch-based* scheduling.

3.1 MapReduce scheduling layers

MapReduce scheduling in Hadoop actually consists of two decoupled layers. The upper layer deals with *allocation* or quantity decisions. The lower layer deals with *assignment* decisions. Specifically, the lower layer attempts to implement the allocation decisions of the upper layer, while taking into consideration a variety of real-world assignment issues not known to the upper layer.

3.1.1 Allocation layer

It is assumed that each host is capable of simultaneously handling some maximum number of Map jobs. These are called Map slots. (A similar statement holds for Reduce phase jobs, but we are not concerned with those in this paper.) Aggregating these Map slots over all the hosts in the cluster, one computes the total number S of Map slots. The role of the allocation layer scheme is to apportion the Map slots among the active Map jobs in some intelligent manner. The CIRCUMFLEX scheme is actually an allocation layer scheduler. CIRCUMFLEX is *fair* in the same sense as FAIR and FLEX: Specifically, given a minimum number m_j of Map slots for job j, the scheme will allocate a number of slots $s_j \geq m_j$, thereby preventing job starvation. (CIRCUMFLEX also respects maximum slot constraints: Given a maximum number M_j of Map slots for job j, the scheme will enforce $s_j \leq M_j$ for each job j. FLEX respects maxima as well.) The *resource* constraint [13] is also respected: $\sum_j s_j \leq S$. (Because CIRCUMFLEX schedules at the subjob level with precedence constraints, we note that all of the above carries over naturally. The minimum for a subjob, for example, is the maximum, over all relevant jobs, of the job minima. The maximum for a subjob is the minimum, over all relevant jobs, of the job maxima. Precedence implies that the number of slots s_j is unambiguously defined.)

3.1.2 Assignment layer

It is this layer that makes the actual assignments of Map job tasks (blocks) to slots, attempting to honor the decisions made at the allocation layer to the extent that this is "reasonable." Host *slaves* report any task completions at *heartbeat* intervals, typically on the order of a few seconds. Such completions free up slots and also incrementally affect the number of slots currently assigned to the various jobs. Bookkeeping then yields an effective ordering of the jobs, from most relatively under allocated to most relatively over allocated. For each currently unassigned slot, the assignment model then finds an "appropriate" task from the most relatively under allocated job that has one. The notion of what is appropriate varies with the version of Hadoop. One example is the locality of the block to the host. In some assignment layer implementations, a non-local block from the best job may be passed over for some period of time in favor of a less appropriate job that has a local block left to scan. This is known as *delay* scheduling [30]. Other affinity issues come into play in different assignment layer implementations.

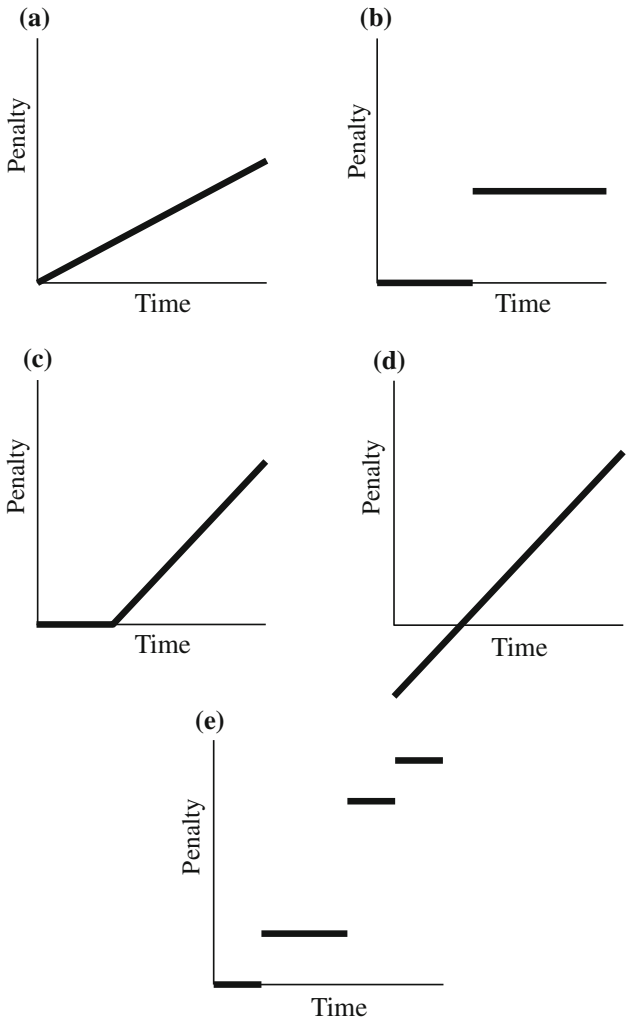

Fig. 5 Per job scheduling metrics. **a** Weighted response time, **b** weighted number of tardy jobs, **c** weighted tardiness, **d** weighted lateness, **e** SLA penality

3.2 Scheduling metrics

Scheduling theory [4,16,19] basically attempts to optimize schedules with respect to certain *metrics* (or *penalty functions*). Given a particular job, a metric measures the "cost" of completing that job at a particular time. The subfigures in Fig. 5 describe the five most common categories of per job

metrics in the scheduling theory literature. Several combinatorial alternatives exist within most of the categories. For example, we will see below that each of the first four categories can be weighted or not. (In some cases, specific weight choices will have special meanings. In other cases, they simply define the relative importance of each job.) Also, one might choose to either minimize the sum of the per job metrics (a *minisum* problem), or minimize the maximum of the per job metrics (a *minimax* problem). Optimizing each of these alternatives serves a different but useful purpose.

3.2.1 Response time

The metric illustrated in Fig. 5a is probably the most commonly employed in computer science. (The weight is the slope of the linear function.) There are several natural examples. Solving the minisum problem effectively minimizes the average response time, weighted, or otherwise, because the sum differs from the average by a multiplicative constant. (Such a constant does not affect the solution to the optimization problem.) In the unweighted case, solving the minimax problem minimizes the *makespan* of the jobs. This is the completion time of the last job to finish and is appropriate for optimizing batch work. Suppose the *work* (which can be defined as the minimum response time required to perform job j in isolation) is W_j. As we have noted, the completion time of a job divided by W_j is known as the *stretch* of the job. It is a measure of how delayed the job will be by having to share the system resources with other jobs. Thus, solving a minisum problem while employing weights $1/W_j$ will minimize the average stretch of the jobs. Similarly, solving a minimax problem while employing weights $1/W_j$ will minimize the maximum stretch. Either of these are excellent *fairness* measures and are in fact more commonly used than either average or maximum PWT, the fairness metrics in AKO.

3.2.2 Number of tardy jobs

Here, each job j has a *deadline*, say D_j. In this case, only the minisum problem is appropriate. The weight is the height of the "step" in Fig. 5b. The unweighted case counts the number of jobs that miss their deadlines, clearly a useful metric. The weighted case counts some jobs more than others.

3.2.3 Tardiness

Again, each job j has a deadline D_j. The tardiness metric generalizes the response time metric, which can be said to employ deadlines at time 0. Only tardy jobs are "charged," and the slope of the non-flat line segment in Fig. 5c is the weight. It makes sense to speak of either minisum or minimax tardiness problems, both either weighted or unweighted.

3.2.4 Lateness

Once again, each job j has a deadline D_j. The lateness metric generalizes response time also. As before, the weight is the slope of the line. Note that "early" jobs are actually rewarded rather than penalized, making this the only potentially negative metric. The minisum variant differs from the response time metric by an additive constant and thus can be solved in exactly the same manner as that problem. But the minimax problem is legitimately interesting in its own right. See Fig. 5d.

3.2.5 SLA costs

In this metric, each job j has potentially multiple pseudo-deadlines $D_{j,k}$ that increase with k. And the penalties $p_{j,k}$ increase with k also. This yields Fig. 5e, a step function for each job, clearly a generalization of the weighted number of tardy jobs metric. As in that case, only the minisum problem is appropriate. One can think of this metric as the total cost charged to the provider based on a prenegotiated *SLA* contract.

3.3 Speedup functions

From a scheduling perspective, a key feature of the Map phase of a MapReduce job is that it is *parallelizable*. Roughly speaking, it is composed of many atomic tasks that are effectively independent of each other and, therefore, can be performed on a relatively arbitrary number of (multiple slots in) multiple hosts simultaneously. If a given job is allocated more of these slots, it will complete in less time. In the case of a Map phase job, these atomic tasks correspond to the blocks. (In the case of a Reduce phase job, the atomic tasks are created on the fly, based on keys.) The CIRCUMFLEX scheme takes advantage of this additional structure inherent in the MapReduce paradigm.

We now describe the relevant scheduling theory concepts formally. See, for example, [16]. Consider a cluster with a total of S homogeneous Map phase slots. (To be concrete, we choose to use MapReduce terminology here rather than the more standard theoretical scheduling terminology. The latter would employ the word *processors* instead of slots.)

A job is said to be *parallel* if it can be performed using some fixed number s (between 1 and S) of slots simultaneously, with an execution time E. (One can think geometrically of this job as a rectangle with width equal to the number of slots s and height equal to the execution time E.) A job is said to be *parallelizable* if it can be performed variously on an arbitrary number $1 \leq s \leq S$ of slots simultaneously, with an execution time $F(s)$ that depends on the number of slots allocated. The execution time function F is known as the *speedup* function. It can be assumed without loss of

generality to be non-increasing, because if $F(s) < F(s+1)$, it would be better to simply leave one slot idle. This would result in a new (replacement) speedup function $\bar{F}$ for which $\bar{F}(s+1) = \bar{F}(s) = F(s)$. One can think, also without loss of generality, of a job that can be performed on a subset $P \subseteq \{1, \ldots, S\}$ of possible allocations of slots as being parallelizable: simply define $F(s) = \min_{\{p \in P | p \leq s\}} F(p)$, where, as per the usual convention, the empty set has minimum ∞. In this sense, parallelizable is a generalization of parallel: A parallel job employing s slots has a speedup function that is infinite before s and constant from s onward.

As we shall describe below, the Map phase of a MapReduce job has, to a good approximation, such a speedup function. Fig. 6a illustrates a possible speedup function.

3.4 Moldable and malleable scheduling

The problem of scheduling multiple parallel jobs in order to minimize some given metric can be visualized as a problem of packing rectangles, with job start times as the decision variables. Parallel scheduling has been studied extensively in the scheduling literature [16]. Problems for most of the natural metrics are NP-hard and thus are often tackled via so-called *approximation algorithms* where possible. These are schemes of polynomial complexity that can be shown to be within a certain factor of optimal. See, for example, [7] for the makespan metric and [21] for the weighted and unweighted response time metrics.

Parallelizable scheduling is a generalization of parallel scheduling. The simplest such variant is known as *moldable* scheduling. Here, the problem is to schedule multiple parallelizable jobs to optimize a given metric. The number of slots is treated as an additional decision variable, but once this allocation is chosen, it cannot be changed during the entire execution of the job. The name comes because the rectangles themselves can be thought of as moldable: Pulling a rectangle wider (that is, giving a job more slots) has the effect of making the rectangle less high (that is, executing the job faster). Figure 6b illustrates a potential choice of slot allocations for a moldable job in a MapReduce context. The top-right vertex in the rectangle corresponds to a point on the speedup function of Fig. 6a. Moldable scheduling problems can sometimes be solved by using the parallel scheduling problem algorithm as a subroutine. See, for example, [23] for makespan and [21] for weighted and unweighted response time. In each of these cases, the approximation factor of the parallel algorithm is retained.

Finally, one can generalize moldable scheduling as well, so-called *malleable* scheduling. Here, the problem is to schedule multiple parallelizable jobs to optimize a given metric, as before. But instead of making a permanent decision as to the slot allocations, each job can proceed in multiple intervals. Different intervals can involve different allocations. Each interval contributes a portion of the total work required to perform the job. Figure 6c illustrates a potential three interval choice of slot allocations for a malleable job in a MapReduce context. See, for example, [5] for makespan.

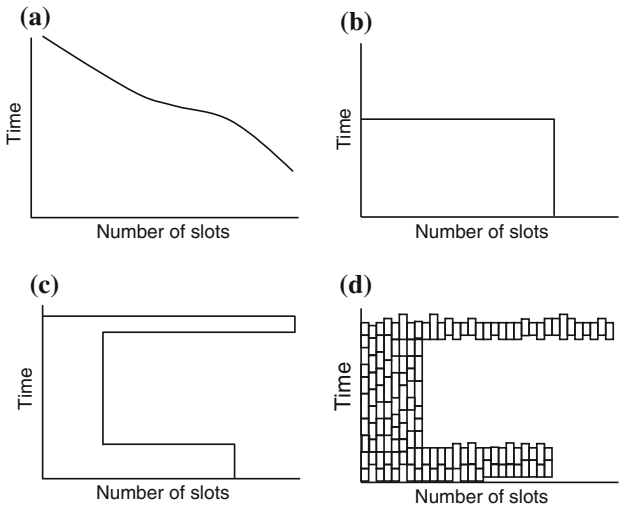

Fig. 6 Speedup functions, moldable and malleable jobs, task assignments. **a** Speedup function, **b** moldable job, **c** malleable job, **d** tasks

The optimal malleable schedule for a particular scheduling problem instance will have a cost less than or equal to that of the optimal moldable schedule for the same problem instance, since moldable schedules are also malleable. In this paper, we are attempting to find an optimal malleable scheduling solution for each of the relevant objective functions. But we will first solve the moldable scheduling problem. The solution there will then help us to solve the more general malleable problem.

3.5 Malleable scheduling and the model

Now, we explore how well the MapReduce model fits the malleable scheduling framework. There are several aspects to the answer.

To begin with, the parallel job structure in MapReduce is a direct consequence of the decomposition into small, independent atomic tasks. Consider the assignment layer. Its role is to approximate at any given time the decisions of the allocation layer by assigning job tasks. Consider Fig. 6d. Over time, the assignment layer will nearly replicate the malleable allocation decisions given in Fig. 6c by a set of job task to slot decisions. The approximation is not perfect, for several reasons: First, slots may not free up at precisely the time predicted by the allocation layer. So, new task slots will not be assigned perfectly: at the bottom of each interval, the rectangle may be slightly jagged. Likewise, slots may not end at precisely the time predicted by the allocation layer. Thus, at the top of each interval, the rectangle may also be slightly

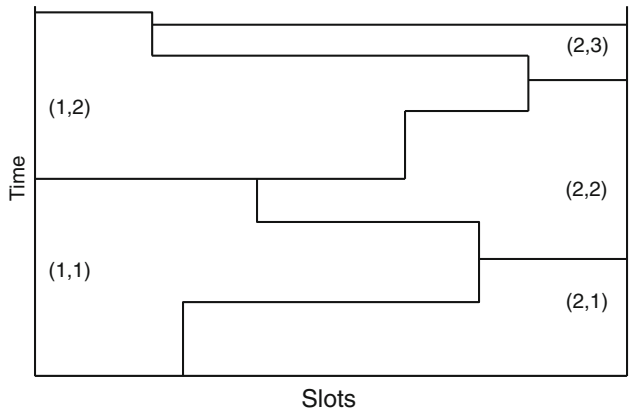

Fig. 7 Chain precedence malleable schedule

jagged. Finally, the assignment layer may relax adherence to the exact allocation goals for the various jobs, for example, in order to ensure host or rack locality [29,30].

But the first two are modest approximations *because* individual task times are small relative to the total time required for the job. And discrepancies between the allocation layer goals and the assignment layer are modest by definition. In summary, as illustrated in Fig. 6c, d, the malleable scheduling model fits the reality of the MapReduce paradigm closely.

Moreover, because the tasks are independent, the total amount of work involved in a job is essentially the sum of the work of the individual tasks. Therefore, in conjunction with the statements above, we can actually assume a speedup function of a very special kind: the speedup should be close to *linear*, meaning a speedup function of the form $F(s) = C/s$ between its minimum and maximum numbers of slots. (Here, C is a constant for a job proportional to the amount of work required to perform that job.) Note that this is not a statement particularly affected by the factors such as locality. Such factors would affect the constant, not the shape of the speedup function, which is a truncated hyperbola. (The truncation is because of the minimum and maximum number of slots for the job. So, as noted before, the speedup function will be infinite for $s < m_j$ and will flatten out at C/M_j for $s \geq M_j$.)

Speedup functions for individual jobs must be estimated in order for CIRCUMFLEX to do its job. Fortunately, by assuming relative uniformity among the job task times, we can continually extrapolate the times for the remaining tasks from the times of those tasks already completed. In this manner, the refined estimates should naturally become more accurate as the job progresses, epoch by epoch. Periodic jobs can also be seeded with speedup functions obtained from past job instances.

3.6 Malleable scheduling with precedence constraints

We note that malleable scheduling can be done in the context of subjobs with precedence constraints as well. Figure 7 illustrates a potential malleable schedule of 2 datasets with a total of 5 subjobs. The 1st dataset has 2 subjobs, namely (1, 1) and (1, 2), related by a precedence constraint. The 2nd dataset has 3 subjobs, (2, 1), (2, 2) and (2, 3), related by chain precedence constraints.

3.7 Epoch-based scheduling

The CIRCUMFLEX scheme is an example of an *epoch*-based allocation scheduler. This means that time is partitioned into epochs of some fixed length T. So, if time starts at $t = 0$, the epochs will start at times 0, T, $2T$, $3T$, and so on, labeled accordingly. The scheduler will produce allocations that will be in effect for one epoch, so that the eth epoch allocations will be honored from time eT to time $(e + 1)T$. Obviously, the work for the eth epoch must be completed by the start time eT of that epoch.

The CIRCUMFLEX scheme receives input describing the total number of Map slots in the system, the number of active Map jobs, their subjobs, the minimum and maximum number of slots per job, the chain precedence constraints, and estimates of the remaining processing times required for each of the subjobs. Then, the algorithm outputs a high-quality malleable schedule consisting of allocations of slots to subjobs in some number of intervals. Allocations for the eth epoch will likely extend beyond the start time of the $(e+1)$st epoch. But all allocation decisions will be superseded by the decisions of the newest epoch. In fact, it is expected that the completion time of even the *first* of the consecutive intervals in the eth epoch will typically exceed the length of an epoch. This means that generally only the first interval in the output will actually be enforced by the assignment model during each epoch.

An advantage of an epoch-based scheme is its resilience over time. Epoch by epoch, the CIRCUMFLEX scheme automatically corrects its solution in light of more current work estimates, newly arrived or departed jobs, and cluster state changes.

4 CIRCUMFLEX scheduling

Assuming the MapReduce and theoretical scheduling preliminaries outlined above, we are in a position to describe the CIRCUMFLEX scheduler. As before, we shall focus on the fully overlapping case, but also briefly describe the semi-shared case where appropriate. In the former case, CIRCUMFLEX is an epoch-based malleable allocation layer scheduler for subjobs related by chain precedence. It eliminates the last two disadvantages noted for AKO. Specifically, it optimizes both average and maximum stretch, plus a number of other metrics more natural than those of AKO. (Of the 16 natural combinatorial choices handled by FLEX, the current version of

CIRCUMFLEX can handle 11.) Additionally, CIRCUMFLEX handles the minimum constraints that are inherent in both FAIR and FLEX.

First, we justify the chain precedence assumption among the subjobs. Then, we describe the two steps of the CIRCUMFLEX scheduler, each of which generalizes FLEX. The first step is to solve one of the two optimization problems, depending on the precise metric chosen. In this step, we wish to find a high-quality (linear) *priority order* for the subjobs. Actually, this priority ordering will also be a topological order of the subjobs. (Recall that in the "job" context, a *topological order* is an ordering of the jobs which respects the precedence among the jobs. Thus, $j_1 < j_2$ whenever $j_1 \prec j_2$.) In one case (minisum average response time variants, specifically average response time, weighted average response time, and average stretch), the optimization problem can be solved by a *Generalized Smith's Rule* scheme. This is, as its name implies, a generalized version of Smith's Rule [19]. In the other case (all minimax variants), it can be solved by a *Backwards Dynamic Programming* scheme. But either of the optimization schemes provide as output a *priority order* of the various subjobs, which is then used as input by the second step. And because of the imposed topological order, the schedule output by the second step will retain the original precedence constraints. In particular, the output of the second step will be an optimized malleable schedule of the subjobs for the chosen metric in the cyclic piggybacking environment. We have designed a new *Ready List Malleable Packing* scheme to solve this second problem, generalizing the scheme first introduced in [26].

4.1 Precedence

Suppose, as before, that there are K_d jobs scanning a given dataset d at a particular instant in time. We have seen that this dataset gives rise to K_d subjobs, namely $\{(d, 1), \ldots, (d, K_d)\}$. Cyclic piggybacking has the effect of partitioning the dataset d into $K_d + 1$ disjoint sets of blocks. The first set is relevant to all K_d jobs. The second set is still relevant to $K_d - 1$ jobs, all but the first to arrive. (The first job has already scanned these blocks.) The third set is still relevant to $K_d - 2$ jobs, all but the first two to arrive. Continuing in this nested manner, the K_dth subset is still relevant to 1 job, the last to arrive. The (K_d+1)st subset, which is empty if and only if the last job has *just* arrived, is no longer relevant. In general, subjob (d, k) is relevant to $K_d - k + 1$ jobs.

We claim that the subjobs associated with each dataset d can be assumed in an optimal schedule to be related by chain precedence. In other words, $(d, 1) \prec (d, 2) \prec \cdots \prec (d, K_j - 1) \prec (d, K_j)$. A simple interchange argument suffices to see this: No actual job can complete until *all* the blocks associated with its dataset have been scanned. And all of the possible scheduling metrics are functions of this completion time. If $1 \leq k_1 < k_2 \leq K_j$, it can help the scheduling objective function to perform the scan of a block in subjob (d, k_1) before performing the scan of a block in subjob (d, k_2). This is because all of the original jobs that are relevant to subjob (d, k_2) are also relevant to subjob (d, k_1). Furthermore, not performing the scans in this order will actually hurt, because it wastes resources that could be dedicated to completing subjob (d, k_1) earlier. So, after interchanging the block scans into the proper order, the result follows. Again, see Fig. 3, where $d = 1$, $K_j = 4$, and we can assume that $(1, 1) \prec (1, 2) \prec (1, 3) \prec (1, 4)$.

In the semi-shared case, the interchange argument yields a more general precedence relationship between the subjobs created. As before, no job can complete until all the blocks associated with it have been scanned. Considering Fig. 4, it can help to scan the three-way intersection before two-way intersections, and the two-way before the one-way. Similarly, failure to proceed in this order will hurt. The resulting precedence graph is shown. (The arrows indicate precedence.) Of the three crucial algorithms of CIRCUMFLEX only the Generalized Smith's Rule breaks down. It requires chain precedence. But the Backwards Dynamic Programming and Ready List Malleable Packing schemes carry over unchanged. So, in the semi-shared scan case, we can handle 8 minimax metrics, but not weighted average response time metrics.

4.2 Finding a priority ordering

For finding a priority ordering, one of the two schemes is employed, depending on the problem variant. The first case handles minisum average response time variants, specifically average response time, weighted average response time, and average stretch. The second case handles all minimax metrics. Either scheme produces an interim schedule that maintains the precedence constraints (optimal for a *single* processor), and the sequencing of the jobs in this interim schedule determines the input ordering to the Ready List Malleable Packing scheme.

4.2.1 Weighted average response time

This case is solved by a generalized version of Smith's Rule [19]. (Smith's Rule optimally solves the problem of minimizing weighted average response time for independent jobs j with weight w_j and processing time p_j on one processor by sequencing the jobs in non-decreasing order of the ratios p_j/w_j.) The pseudo-code for the Generalized Smith's Rule is given in Fig. 8. It will be clear that this does, indeed, represent a generalization of Smith's Rule to the case of chain precedence subjobs. See line 7, in particular, where the partial ratios of sums replace the traditional Smith Rule ratios. The proper ordering is achieved via line 8. Since chain precedence is maintained in the resulting sequence of subjobs,

1: **for** $d = 1$ to D **do**
2: Create ordered chain of subjobs $\mathcal{L}_d = \{(d,1),...,(d,K_d^1)\}$ for dataset d, where (d,K_d^1) represents the last subjob for dataset d, and where subjob (d,k) has processing time $p_{d,k}$ and weight $w_{d,k}$.
3: Set $K_d^0 = 1$
4: **end for**
5: Set $\mathcal{L} = \cup_{d=1}^{D} \mathcal{L}_d$
6: **while** $\mathcal{L} \neq \emptyset$ **do**
7: Compute for each subjob (d,κ) with κ between K_d^0 and K_d^1 the partial ratio of sums $\sum_{k=K_d^0}^{\kappa} p_{d,k} / \sum_{k=K_d^0}^{\kappa} w_{d,k}$
8: Find subjob $(\bar{d},\bar{\kappa})$ for which the partial ratio of sums is minimized
9: Schedule subjobs $(\bar{d},K_d^0),...,(\bar{d},\bar{\kappa})$
10: Set $\mathcal{L} = \mathcal{L} - \{(\bar{d},K_d^0),...,(\bar{d},\bar{\kappa})\}$
11: Update $K_{\bar{d}}^1 = \bar{\kappa} + 1$
12: **end while**

Fig. 8 Generalized Smith's Rule

1: Set $\mathcal{J} = \{1,...,J\}$
2: Compute the subset $\mathcal{J}' = \{j \in \mathcal{J} | \nexists j' \in \mathcal{J}, j \prec j'\}$
3: **while** $\mathcal{J}' \neq \emptyset$ **do**
4: Compute $j^* = arg\min_{j \in \mathcal{J}'} F_j(\sum_{j \in \mathcal{J}} p_j)$
5: Remove j^* from $\mathcal{J}$
6: Recompute $\mathcal{J}'$
7: **end while**

Fig. 9 Backwards Dynamic Programming

the ordering of the completion times of the subjobs is a topological ordering as well, and this priority ordering is input to the second step.

Unfortunately, the Generalized Smith's Rule requires chain precedence and is not suitable for the case of semi-shared scans.

4.2.2 Minimax problems

This case is solved by a Backwards Dynamic Programming. The pseudo-code for this case is given in Fig. 9. This single processor scheme actually works for arbitrary jobs, any non-decreasing penalty function F_j, and any precedence relation $\prec$. *Chain* precedence is not required. So, we have described the pseudo-code more generically. As the name implies, the dynamic program schedules the jobs in reverse order, from last to first. (See line 4 for the selection criteria.) Since precedence is maintained in the resulting sequence of subjobs, the ordering of the completion times is again a topological ordering, and this priority ordering is input to the second step.

4.3 Ready List Malleable Packing scheme

The second step again works for arbitrary precedence constraints, so we describe it in generic job terminology. The translation to subjobs and chain precedence is easy. This algorithm is a generalization of the one first presented in [26].

1: Set time $T_0 = 0$
2: Create list $\mathcal{L} = \{1,\ldots,J\}$ of jobs, ordered by priority and respecting precedence
3: Create sublist L_1 of ready jobs
4: **for** $i = 1$ to J **do**
5: Allocate m_j slots to each job $j \in L_1$
6: Set $L_2 = L_1$, with implied ordering
7: Compute slack $s = S - \sum_{j \in L} m_j$
8: **while** $s > 0$ **do**
9: Allocate $\min(s, M_j - m_j)$ additional slots to highest priority job $j \in L_2$
10: Set $L_2 = L_2 \setminus \{j\}$
11: Set $s = s - \min(s, M_j - m_j)$
12: **end while**
13: Find first job $j \in L_1$ to complete under given allocations
14: Set T_i to be the completion time of job j
15: Set $L_1 = L_1 \setminus \{j\}$
16: Add all immediate successor jobs to j in ready list L_1
17: Compute remaining work for jobs in $\mathcal{L}$ after time T_i
18: **end for**

Fig. 10 Ready List Malleable Packing scheme

The scheme inputs one of the output priority orderings from the previous subsection, as appropriate. It then employs *Ready List Malleable Packing scheme*. (A *ready list* is a dynamically maintained list of jobs that are ready to run at any given time. In other words, all precedence constraints must have been satisfied at the time a job is put on the list.)

The pseudo-code for the Ready List Malleable Packing scheme appears in Fig. 10. Given a priority ordering, the scheme proceeds iteratively. At any iteration, a *current* list $\mathcal{L}$ of jobs is maintained, ordered by priority. Time is initialized to $T_0 = 0$. The current list $\mathcal{L}$ is initialized to be all of the jobs, and one job is removed from $\mathcal{L}$ at the completion time T_i of each iteration i. Call the time interval during iteration i (from time T_{i-1} to T_i) an *interval*. The number of slots allocated to a given job may vary from interval to interval, thus producing a malleable schedule.

The ith iteration of the algorithm (lines 4 through 18) involves the following steps: First, the scheme allocates the minimum number m_j of slots to each job $j \in \mathcal{L}$. This is feasible, since the minima have been normalized, if necessary, during a precomputation step. After allocating these minima, some slack may remain. This slack can be computed as $s = S - \sum_{j \in L} m_j$. The idea is to allocate the remaining allowable slots $M_j - m_j$ to the jobs j in priority order. The first several may get their full allocations, and those jobs are allocated their maximum number of slots, namely $M_j = m_j + (M_j - m_j)$. But ultimately, all S slots may get allocated in this manner, leaving at most one job with a "partial" remaining allocation of slots, and all jobs having lower priority with only their original, minimum number of slots. (The formal details of these steps are given in the pseudo-code.) Given this set of job allocations, one of the jobs j completes first, at time T_i. (Ties among jobs may be adjudicated in priority order.) Now, job j is removed from $\mathcal{L}$, and the necessary bookkeeping is performed to compute the remaining work

past time T_i for those jobs remaining in $\mathcal{L}$. After J iterations (and J intervals), the list $\mathcal{L}$ is depleted and the output malleable schedule created.

Note that the ready list construct ensures that this scheme produces a malleable schedule which respects the precedence constraints.

4.4 Algorithmic complexity

Each of the three CIRCUMFLEX scheduling algorithms happens to have complexity $O(J^2)$. This includes the Generalized Smith's Rule, Backwards Dynamic Programming, and the Ready List Malleable Packing scheme. Since we are solving precisely two of these problems once per epoch, we believe CIRCUMFLEX will be sufficiently fast to handle reasonably large problem instances. However, truly large problems should still be readily tractable via divide and conquer decomposition techniques.

5 CIRCUMFLEX implementation

We support circular scans in MapReduce by translating each subjob into a MapReduce job that uses modified Map and Reduce tasks, called C-mappers and C-reducers, respectively. C-mappers identify all Map functions that need to be executed by the subjob and proceed to call every function on every input record. They prefix every output record with the job ID of the Map function that produced it. The outputs are partitioned by this job ID first. Within each partition, a C-mapper uses the comparison and partitioning functions of the original jobs. C-reducers are modified to read not only the results of the C-mappers of this subjob, but also of all the previous subjobs of the MapReduce job in question.

For example, `Subjob1` in Fig. 11 runs Map functions of three jobs. It produces all output partitions of `Job1` followed by partitions `Job2` and then `Job3`. C-reducers of `Subjob1` read only the output partitions of `Job1`. C-reducers of `Subjob2` read partitions of `Job2` from both `Subjob1` and `Subjob2`. Finally, C-reducers of `Subjob3` read output partitions of `Job3` from all three subjobs.

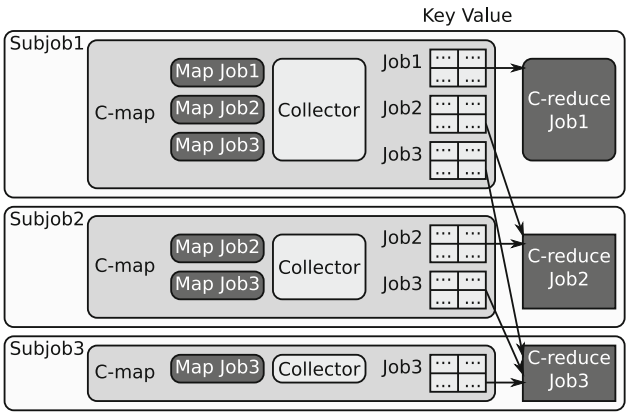

Fig. 11 Shared scans in MapReduce

In Hadoop, the dominant open-source MapReduce implementation today, C-mappers can be implemented utilizing the existing APIs and without modifying the Hadoop framework itself. C-reducers, in contrast, require modifications to Hadoop reducer code. Special care must be taken to preserve the fault tolerance of Hadoop, since C-reducers break a key assumption of Hadoop by introducing dependencies between Hadoop jobs.

In order to facilitate communication between job clients, C-mappers and C-reducers of multiple subjobs, we utilize Apache ZooKeeper, an open-source distributed coordination service. We first recall some background about Hadoop and ZooKeeper and then describe the architecture of C-mappers and C-reducers.

5.1 MapReduce and Hadoop

In the open-source community, Hadoop [10] is a popular implementation of the MapReduce paradigm. Data are initially partitioned across the nodes of a cluster and stored in a distributed file system (DFS). Data are represented as `(key, value)` pairs. The computation is expressed using two functions:

```
Map (k1,v1) → list(k2,v2);
Reduce (k2,list(v2)) → list(k3,v3).
```

Figure 12 shows the data flow in a MapReduce computation. The computation starts with a Map phase in which the `map` functions are applied in parallel on different partitions of the input data, called splits. A Map task, or mapper, is started for every split, and it iterates over all the input `(key, value)` pairs applying the Map function. The `(key, value)` pairs output by each mapper are assigned a partition number based on the `key` and sorted by their partition number and the `key` using a fixed-size memory buffer. Once the buffer is filled up, the sorted run is spilled to the local disk. At the end of the mapper execution, all the spills are merged into a single sorted file and sent across the cluster

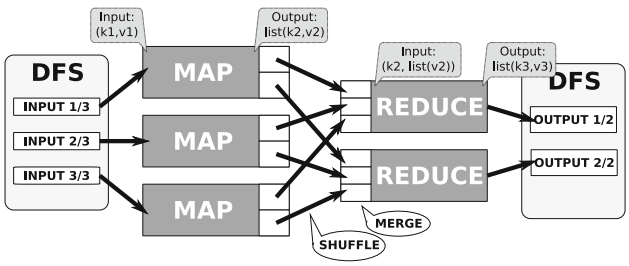

Fig. 12 Data flow in a MapReduce computation

in a shuffle phase. At each receiving node, a reduce task, or reducer, fetches all of its sorted partitions during the shuffle phase and merges them into a single sorted stream. All the pair values that share a certain key are passed to a single reduce call. The output of each `reduce` function is written to a distributed file in the DFS.

Finally, the framework also allows the user to provide initialization and teardown functions for each MapReduce function and customized hashing and comparison functions to be used when partitioning and sorting the keys.

5.2 ZooKeeper

CIRCUMFLEX needs to synchronize multiple Hadoop clients that independently submit jobs for the same dataset into a single circular scan. To this end, we need a distributed metadata store that can perform efficient distributed reads and writes of small amounts of data in a transactional manner. The Hadoop project includes just such a tool, a distributed coordination service called Apache ZooKeeper [2,12]. ZooKeeper is highly available, if configured with three or more servers, and fault tolerant. Data are organized in a hierarchical structure similar to a file system, except that each node can contain both data and subnodes. A node's content is a sequence of bytes and has a version number attached to it. A ZooKeeper server keeps the entire structure and the associated data cached in memory. Reads are extremely fast, but writes are slightly slower because the data need to be serialized to disk and agreed upon by the majority of the servers. Transactions are supported by versioning the data. The service provides a basic set of primitives, like `create`, `delete`, `exists`, `get`, and `set`, which can be easily used to build more complex services such as synchronization and leader election. Clients can connect to any of the servers and, in case the server fails, they can reconnect to any other server while preserving sequential consistency. Moreover, clients can set watches on certain ZooKeeper nodes and get a notification if there are any changes to those nodes.

5.3 C-mappers and C-reducers

The C-map and C-reduce tasks get set up by job clients that all connect to a single ZooKeeper service. ZooKeeper data structure contains a node per dataset. When the client sets up a job j that reads a dataset d, it locates this datasets's node Z_d in ZooKeeper. First, it creates a 'lock' subnode of Z_d and does not proceed until this write succeeds. This ensures that all job clients that want to read the dataset are serialized during the following critical section.

For every data split d_i of the dataset d, the client reads the corresponding split data structure from the ZooKeeper node Z_{d_i}, a child of Z_d. If node Z_{d_i} exists in ZooKeeper, the client appends j's ID to the job list in the node—these splits will be executed by the already existing subjobs. They basically "ride on the bus" that is already scheduled to leave the "station," so we refer to them as *rider* splits. If Z_{d_i} does not exist, the client creates a new node Z_{d_i}, with only j's ID in the job list—these splits form the new subjob for j, and we call them *drivers*. This critical section is fast, usually subsecond, so performance impact of serialization is negligible.

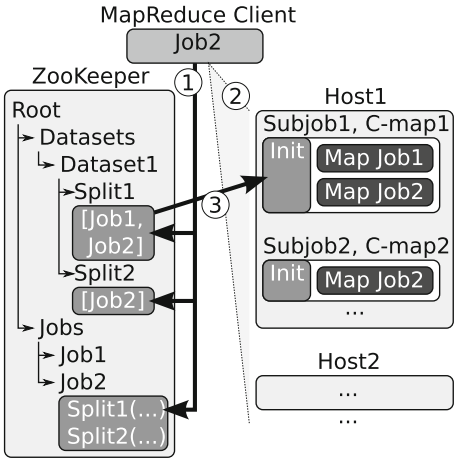

Fig. 13 MapReduce client (*1*) updates the data structure in ZooKeeper, and (*2*) sets up C-map tasks, which in turn (*3*) read ZooKeeper

For example, Fig. 13 shows a job client of `Job2` that needs to read `Dataset1`, which consists of two splits. The client reads the ZooKeeper structure for `Split1` and appends `Job2`. For `Split2`, the structure does not exist, so the client creates it. Thus, for `Job2`, `Split1` is a rider and `Split2` is a driver.

For every rider split, the job client computes its reduce partition offset—the sum of the number of partitions for all the jobs before j in that split's job list. Recall that a C-mapper produces output partitions of all the jobs in its job list, in order. For example, `Subjob1` in Fig. 11 produces all output partitions of `Job1`, followed by partitions `Job2` and then `Job3`. Thus, the partition numbers of `Job1` are unchanged. The partition numbers of `Job2` are shifted by a fixed offset—the number of partitions of `Job1`. These output partitions will be read by C-reducers of `Subjob2`, not `Subjob1`. For `Job3`, the offset is the sum of partition counts of `Job1` and `Job2`.

The offset information is needed by both C-mappers (for their partitioner functions) and C-reducers (to know which partitions of previous subjobs to read). To store this information, the job client creates a single node in ZooKeeper that corresponds to job j. This node contains a table of splits with their offset for partitions of job j, and the job ID of the driver, the first job in the job list. This *offset table* is stored in a compact form, since all splits in a subjob share the job lists and the offsets.

The client submits a job to MapReduce that has a C-map task for every driver split. If there are no driver splits, the job contains a single NO-OP Map task needed to start the reducers. When a C-mapper starts, it reads the ZooKeeper node that corresponds to its input split, and deletes this node, making sure that it deletes the same version that it just read. (Otherwise, it is read again.) This atomic read-and-delete marks the cutoff when the bus leaves the station, so no more jobs are allowed to ride. The C-mapper takes the job list from its ZooKeeper node and proceeds to read the job configurations of all these jobs, combining all their Map tasks into one.

The high-level architecture of a C-mapper is shown in Fig. 11. A C-mapper runs initialization functions of its Map tasks. Then, it reads the input records, for each record sequentially invoking the Map function of every Map job. Each Map function is given a different output collector, which acts as a wrapper over the real output collector and prefixes every output key with the job ID of the Map function producing the output. Similarly, the C-mapper contains a custom comparator that unwraps the output key and calls the comparator of the corresponding job. The C-mapper partitioner function unwraps the key, calls the corresponding partitioner function, and also reads from ZooKeeper the job's partition offset within the split. The final partition number is obtained by adding this offset and the partition number returned by the job's partitioner. Finally, the C-mapper runs the teardown functions of its Map tasks. C-reducers start by reading their job's offset table from the ZooKeeper. For every split, the table contains the driver job ID j_d and a partition offset O. Normally, a reducer number N of a job j reads N's partition from the output of every Map task of j. C-reducers also read partition $O+N$ of j_d's output, for every split in the offset table.

In case of a system failure, this output partition may not exist, and since job j has already finished, Hadoop cannot re-execute its Map tasks as it normally does in case of failures. To work around this problem, C-reducer reports to the framework that one C-mapper in its own job has failed and needs to be re-executed, and puts the information about which split needs to be rerun in ZooKeeper. Once the C-mapper is restarted, it will find the split meta-data in ZooKeeper and read that split instead of its assigned one.

Once all C-reduce tasks are complete, a customized clean-up task removes the job's offset table and other job-related structures from the ZooKeeper.

6 Experiments

6.1 Simulation experiments

In this section, we describe a variety of simulation experiments designed to show the performance of CIRCUMFLEX. In the interests of space, we concentrate on average response time, average stretch, and maximum stretch. The CIRCUMFLEX scheme can, as noted, handle other metrics as well. In the interests of space, we have chosen maximum stretch as representative and most important of the minimax metrics, but maximum tardiness or lateness runs yield comparable results.

We compare FLEX and CIRCUMFLEX with a BATCH variant of our own design. (Recall that the AKO batching schemes described in [1] employed the PWT metric and are basically off-line algorithms.) To compare batching fairly with FLEX and CIRCUMFLEX, we have devised a scheme that batches every dataset scan at the end of a fixed time interval W and then combined this with a FLEX scheduling algorithm applied to the resulting batches. The start times of the intervals are offset evenly, depending on the dataset involved, to space the batch arrivals as equally as possible. This BATCH scheme seems to be in the spirit of AKO. On the negative side, its batching decisions are not as intelligent. On the positive side, it also optimizes the chosen metrics quite well.

The experimental design is as follows. Each experiment simulated 1000 arrivals for a total of $D = 20$ distinct datasets. The popularity of each dataset is chosen by sampling from the CDF of a Zipf-like distribution with parameter θ_1 equal to either 0, .25, .5, .75, or 1.0. (Zipf-like distributions [15] on a set of size D employ a parameter θ between 0 and 1. When $\theta = 1$, the distribution is purely Zipf, and when $\theta = 0$, the distribution is uniform. Zipf-like distributions thus span a wide variety of common skew patterns.) The arrival times themselves are chosen according to a Poisson distribution. The size of each dataset is chosen from a second Zipf-like distribution with parameter θ_2 equal to either 0, .25, .5, .75, or 1.0. The dataset popularity and size Zipf-like distributions are then positively or negatively correlated using a simple scheme. If the correlation is perfectly positive, the largest datasets are also the most popular. If the correlation is perfectly negative, the smallest datasets are most popular. If the correlation is zero, there is no relation between dataset size and popularity. We use 9 different parametric choices of correlation. Combined with the 5 choices for each of θ_1 and θ_2, this gives us excellent coverage, with 225 parametric alternatives. We assumed a total of $S = 100$ Map slots, corresponding to a cluster of 25 nodes if there were 4 Map slots per node. We ran each experiment for a nominal total of $T = 100$ min, though we allowed the Map work to quiesce past this time. Finally, we scaled the dataset sizes so that the total Map times in a non-shared scenario corresponded to 90 % utilization. (This utilization is the total time spent by the Map work divided by the product TS.)

In the experiments, we computed new schedules for each of the alternative schemes upon each new arrival. These schedules were then followed precisely until the next arrival. Metrics for each arrival were computed, of course, based on the difference between the arrival and completion times.

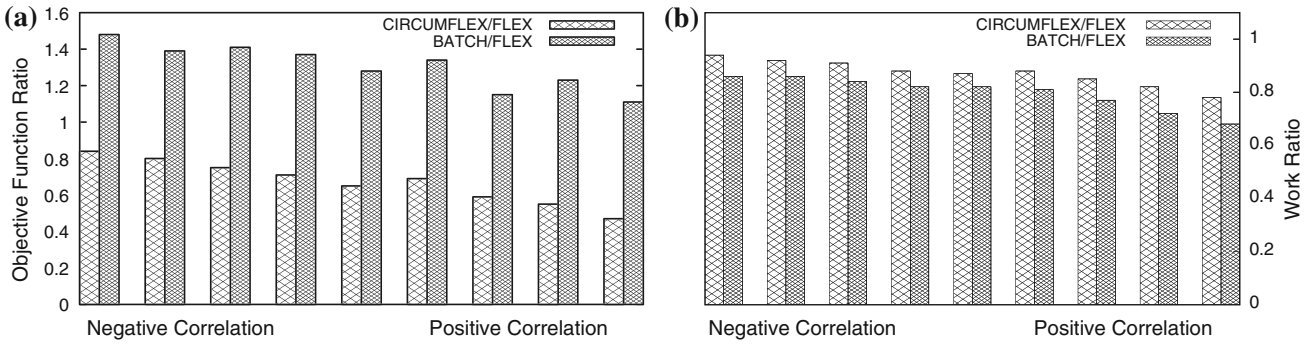

Fig. 14 Average response time ratios, $\theta_1 = \theta_2 = 1$. **a** Objective function ratio, **b** work ratio

We ran 10 repetitions of each experiment, taking averages and standard deviations. We recorded the ratio of the objective function value obtained using CIRCUMFLEX to that of FLEX and similarly that of BATCH to FLEX. We also computed the ratio of the total work for CIRCUMFLEX and BATCH to that of FLEX. For CIRCUMFLEX, we averaged the maximum number of concurrent subjobs over all datasets, and for BATCH we averaged the maximum number of jobs batched together over all datasets. We used batching intervals per dataset of $W = 2$ min (yielding 50 or 51 such intervals in 100 min) as a base case. Note that this adds an average latency of 1 min to each arrival.

Figure 14a shows the average response time ratios of CIRCUMFLEX and BATCH to FLEX for the highly skewed case $\theta_1 = \theta_2 = 1$. Note that the performance of CIRCUMFLEX improves as the correlation ranges from perfectly negative to perfectly correlated. Compared with FLEX, which is optimizing average response time as well, CIRCUMFLEX average response times decrease from 81 % down to 47 %. There is reason to believe that dataset popularity and size might sometimes be negatively correlated, because, for example, many MapReduce jobs might involve the most recent day of data. But even in this case, CIRCUMFLEX performs 19 % better than FLEX. (In the absence of shared scans, FLEX does very well on all metrics, compared to the performance of FAIR and FIFO [26].) On the other hand, BATCH always performs worse than FLEX, because of the trade-offs involved. In the case of perfectly negative correlation, BATCH is 48 % worse than FLEX. In the perfectly positive correlation case, BATCH is still 11 % worse. Standard deviations of these ratios (not shown) are very modest for CIRCUMFLEX, relatively less so for BATCH. This is an indication of the robustness of the CIRCUMFLEX scheme. Figure 14b shows the comparable work ratios for this same example. Both ratios generally decrease from left to right, as one would expect. As expected, CIRCUMFLEX does more work than BATCH but less work than FLEX. The maximum number of concurrent CIRCUMFLEX subjobs averaged 2.8 in the most negatively correlated case and 4.3 in the most positively correlated case. The maximum number of jobs batched together ranged from 4.8 to 5.0, also indicative of the greater efficiency of BATCH.

Figure 15a shows the ranges of the CIRCUMFLEX-to-FLEX ratio and the BATCH-to-FLEX ratio for all 25 parametric choices of θ_1 and θ_2. These are arranged in 5 "planes" of 5 values each. The leftmost group of 5 all correspond to $\theta_1 = 0$ and individually to $\theta_2 = .25 * \tau$, where τ ranges from 0 to 4. (Thus, the detailed data in Fig. 14a are shown in summary form in the rightmost pair of bars in Fig. 15a: the minimum, median, and maximum value across all 9 correlation parameters are shown for each objective function ratio, for both CIRCUMFLEX and BATCH. Note that this is possible to show *because* the ratios never overlap. The worst CIRCUMFLEX ratio is always better than the best BATCH ratio for any particular choice of θ_1 and θ_2. Indeed, the worst CIRCUMFLEX-to-FLEX ratio (.98) in the entire graph is essentially identical to the best BATCH-to-FLEX ratio overall (.97). It is also clear that the ratios for CIRCUMFLEX are much more consistent and tightly clustered across the parametric choices than the ratios for BATCH. CIRCUMFLEX always performs better than FLEX, and BATCH nearly always performs worse than FLEX: The extra average latency of 1 min is too great an impediment for BATCH to overcome.

Figure 15b, c illustrate the corresponding results for average and maximum stretch, respectively. (For maximum stretch, the metric is calculated as the largest value over all 1,000 arrivals.) In both of these metrics, the relative performance of CIRCUMFLEX is even stronger than it is for average response time. A small dataset scan that arrives early in a batching interval causes a high average stretch value and a *very* high maximum stretch value. Both the average and maximum stretch ratios vary widely, depending on the parametric choices. They are never nearly as good as FLEX. On the other hand, the performance of CIRCUMFLEX in both cases is extremely predictable and always much better than that of FLEX. In the case of maximum stretch, the performance differences are dramatic.

We also ran experiments for the average stretch ratios of CIRCUMFLEX and BATCH to FLEX for the modestly skewed

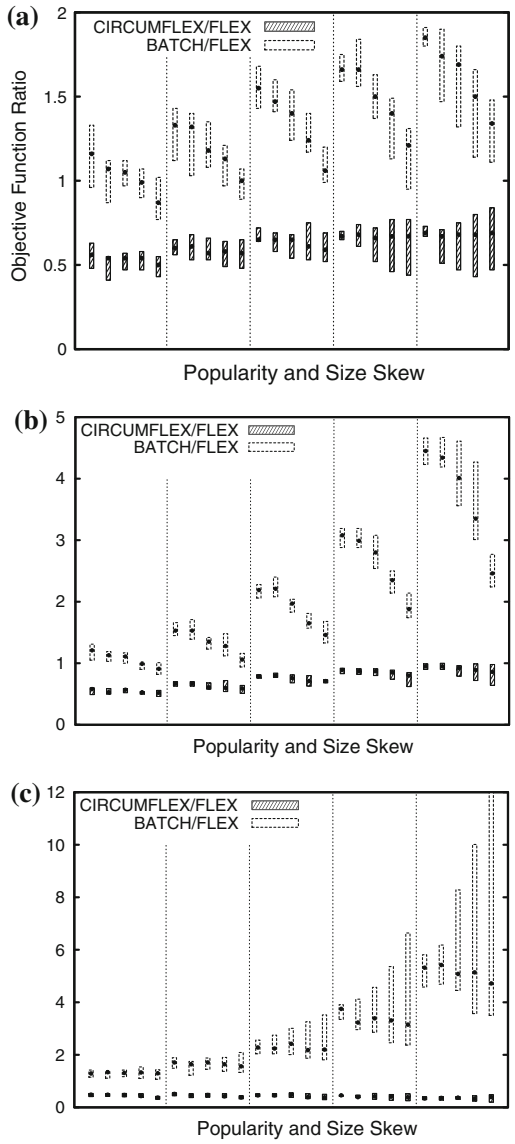

Fig. 15 Ratio summaries. **a** Average response time, **b** average stretch, **c** maximum stretch

case $\theta_1 = \theta_2 = .5$. Here, CIRCUMFLEX improves much more modestly with correlation, from 81 % of FLEX on the left to 68 % on the right. Standard deviations remain quite low, because the CIRCUMFLEX scheme is highly robust. But the BATCH-to-FLEX average stretch ratio is essentially indifferent to the correlation parameter and has a high standard deviation. Moreover, the ratio is itself quite high, with BATCH between 83 and 103 % worse than FLEX. Work ratios (not shown) are between 86 and 92 % for CIRCUMFLEX and between 78 and 85 % for BATCH.

Consider the maximum stretch ratios in the somewhat more skewed case $\theta_1 = \theta_2 = .75$. The metric produces CIRCUMFLEX to FLEX of 89 % in the perfectly negatively correlated case and 74 % in the perfectly positively correlated case. But the ratio of BATCH to FLEX actually increases from 214 to 250 %. This seems to indicate that BATCH is not a good scheme for the maximum stretch metric, unless efficiency is of paramount concern. The work ratios (not shown) for BATCH to FLEX decrease from 84 to 72 %. But even here, the slightly less efficient (92–82 %) CIRCUMFLEX scheme has none of the negative performance properties of BATCH.

While we have not shown all base case experiments in the interest of space, these figures are quite representative. For example, in the case of average response time, in 225 experiments, the best average CIRCUMFLEX-to-FLEX ratio was 41 %, and the worst was 84 %. In the same experiments, the best average BATCH-to-FLEX ratio was 87 %, and the worst was 191 %. Thus, CIRCUMFLEX performs better than FLEX (and better than BATCH) in all cases. By contrast, BATCH performs better than FLEX in 20 (8 %) of the cases. Similar statements are true for both stretch metrics.

The scheme in [1] made the reasonable simplifying assumption that nearly all the work of the Map phase is in the scanning, so that the subsequent computational work can be ignored. We choose the same as a base case for the cleanest possible exposition of the benefits of CIRCUMFLEX. Our simulation experiments with varying the fraction of Map phase work which is due to the scans behave entirely in the expected manner, so we do not show them.

6.2 Benchmark experiments

To evaluate the effectiveness of CIRCUMFLEX in a real-world setting, we performed experiments for the case where there is only one dataset. (Recall that we used twenty in our simulations.) Employing just one dataset allows to ignore the effects of the scheduler, because there are no scheduling decisions to be made. This, in turn, allows us to focus directly on the comparison of cyclic piggybacking with batching.

Our workloads were constructed from a set of queries of the Star Schema Benchmark (SSB) [18], running over a one TB scale SSB dataset. All the experiments were executed on a 42-node IBM iDataPlex cluster. Each node has 8 cores, 32GB of RAM, and 5 disks. Our Hadoop v0.21 used 1 master and 40 slave nodes. The last node was used for clients submitting the queries.

6.2.1 Clydesdale cyclic piggybacking implementation

We executed the queries using Clydesdale [14], a research prototype for structured data processing on Hadoop. Clydesdale achieves dramatic performance improvements over existing solutions without any changes to the underlying MapReduce implementation. To the best of our knowledge, Clydesdale is the fastest solution for processing workloads on structured datasets that fit a star schema on Hadoop. For example, on the Star Schema Benchmark, Clydesdale is on average 38 times faster than Apache Hive [22], the dominant

approach for structured data processing on Hadoop today. Clydesdale achieves this through a novel synthesis of several techniques from the database literature, such as columnar data storage [9], specialized star-join operators, and hash-based local aggregation [24]. Each of these were carefully adapted to the Hadoop environment.

The version of Clydesdale that we used [3] further improves query response time by bypassing the Hadoop job submission mechanism. Accordingly, Clydesdale achieves maximum efficiency, and the bulk of the processing time is actually spent scanning the data. This makes Clydesdale a perfect target for shared scans, either through cyclic piggybacking or batching. We have observed a throughput improvement of up to 3 times for cyclic piggybacking, as compared with sequential query submission. Improvements in average response times were even higher.

The Clydesdale query processor is started by submitting a single Map-only Hadoop job. This starts the worker processes, implemented as Hadoop mappers. The queries are submitted through ZooKeeper. Every worker gets the query information from ZooKeeper, processes the query on its splits of the dataset, and sends the results of local aggregation to the client. Upon query completion, the Hadoop job does not terminate. The workers simply stay idle, waiting for the next query.

Since Clydesdale does not use all of Hadoop's mechanisms, we had to adapt CIRCUMFLEX implementation described in Sect. 5 appropriately. In particular, the C-reducer implementation had to be modified to run in Clydesdale's client module, though its ZooKeeper-based communication module remained largely the same. All the C-maps that access the same dataset now run as different threads of the same Hadoop Map task (or Clydesdale worker).

Normally, Clydesdale shared scans operate in cyclic piggybacking mode, where each worker goes through all of its splits in the same order. The queries arrive at any time and leave once they have been processed on all the splits. To compare CIRCUMFLEX with various batching schemes, we have also implemented a batch mode, where Clydesdale workers execute only the set of queries that they find upon initialization. These batch workers scan every split exactly once and terminate once the initial set of queries is executed.

The query response time is measured on the client as the difference between the query completion time (obtaining the entire final result) and the query submission time. Thus, response time includes transmission of the query to the workers, through ZooKeeper, setting up per-query structures, scanning and executing the query on each worker, transmitting the results of local aggregation back to the client, and performing the final aggregation of the local aggregates.

The best possible response time for an average query, as obtained by running on an idle Clydesdale system, is about 9 s. The bulk of that time (over 6 s) is spent scanning the splits and executing the query's Map function for every record. All other stages of the query execution are typically subsecond. In the next section, we will see that CIRCUMFLEX approaches this minimum response time as interarrival times increase.

6.2.2 Experiment results

We compare average response time and makespan (which is inversely related to throughput) of Clydesdale running CIRCUMFLEX with two types of batching schemes: time-based and size-based. The time-based scheme launches a Clydesdale execution every W seconds, assuming at least one query has arrived. The size-based scheme launches an execution when N queries have arrived.

Since Clydesdale can only efficiently run one batch at a time, we improve batching performance by introducing a heuristic that does not start a batch until the previous one is completed. When a batch starts, it will execute all the jobs that have already arrived and have not yet been executed. It is easy to see that this heuristic is guaranteed not to increase average response time or makespan. In reality, it improves them substantially.

The experimental design is as follows. Each experiment consisted of running the same set of 24 SSB queries, arriving at random times chosen according to a Poisson distribution. We varied the average interarrival time from .5 to 10 s. We repeated every experiment with 5 different seeds for the arrival time random number generator and report the average of these 5 runs.

Figure 16a, b compare average response time and makespan (wall clock time) for CIRCUMFLEX experiments (marked `CFLEX`) with that of a few representative variations of the batching scheme and with sequential query execution (marked `NOSHARE`). It is easy to see that if queries arrive closely together, batching (especially with large batch sizes) is much better than sequential execution. However, if queries are spaced out such that there is little overlap, sequential execution is better, especially in terms of response time.

In Fig. 16a, we notice that SIZE-12 performs better than SIZE-1 for small interarrival times, as expected. However, as the interarrival times increases to about 1.5 s, SIZE-1 catches up with SIZE-12 and soon dominates it significantly. An optimal batch size would depend on the interarrival time, and in fact, from Fig. 17a, we see that a batch size of 4 performs best for interarrival times up to 5 s. In all cases, CIRCUMFLEX outperfoms size-based batching, sometimes even by a factor of 3.

CIRCUMFLEX is clearly and uniformly better than all of the alternatives. If there were exactly zero overlap between query executions, CIRCUMFLEX and sequential execution would have the same performance. However, given Poisson interarrival times, even with 10 s averages, some queries will arrive closely together. Thus, CIRCUMFLEX performs better even in such cases.

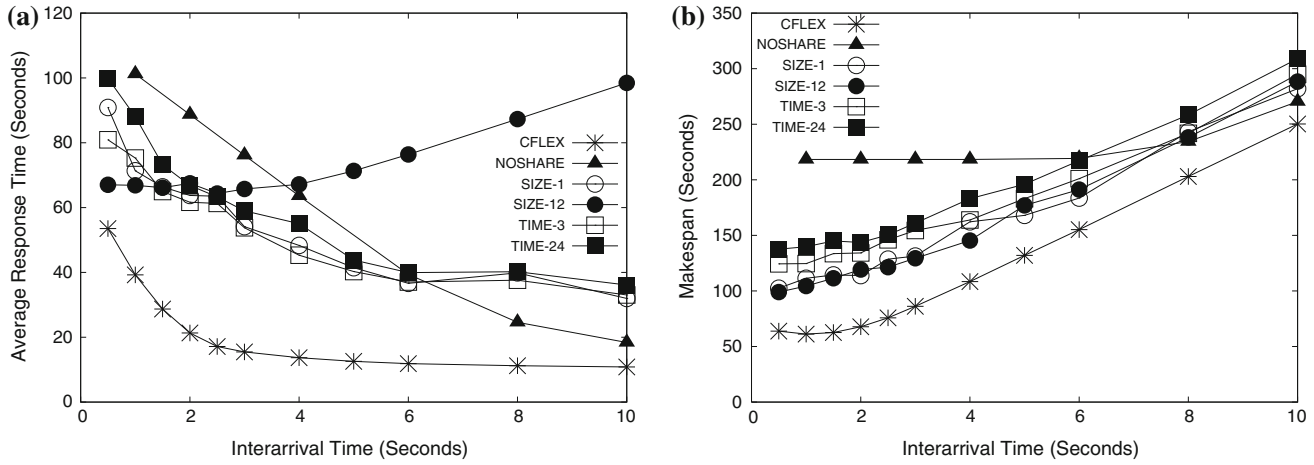

Fig. 16 CIRCUMFLEX versus sequential execution versus batching. **a** Average response time, **b** makespan

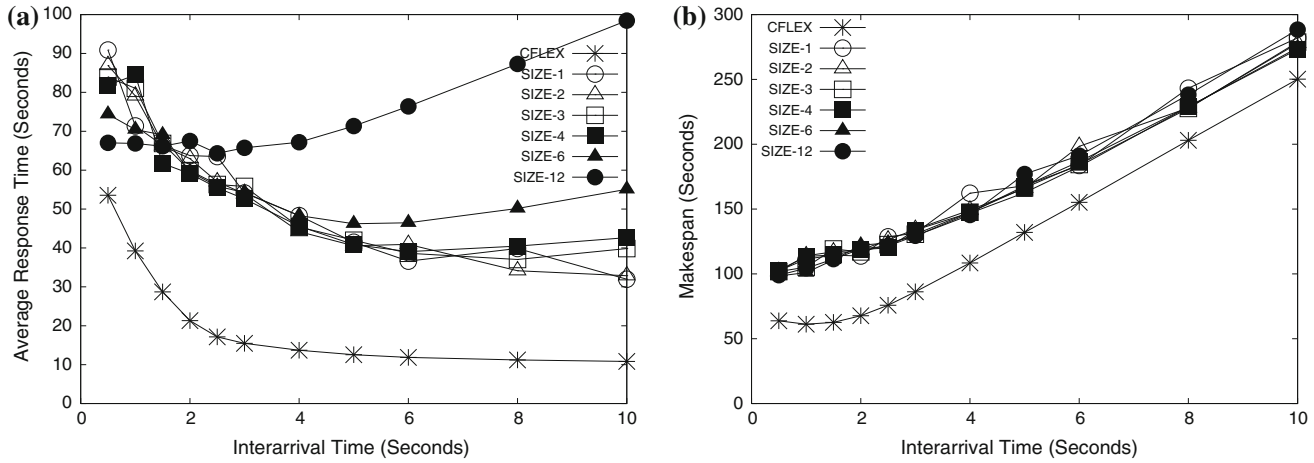

Fig. 17 CIRCUMFLEX versus size-based batching. **a** Average response time, **b** makespan

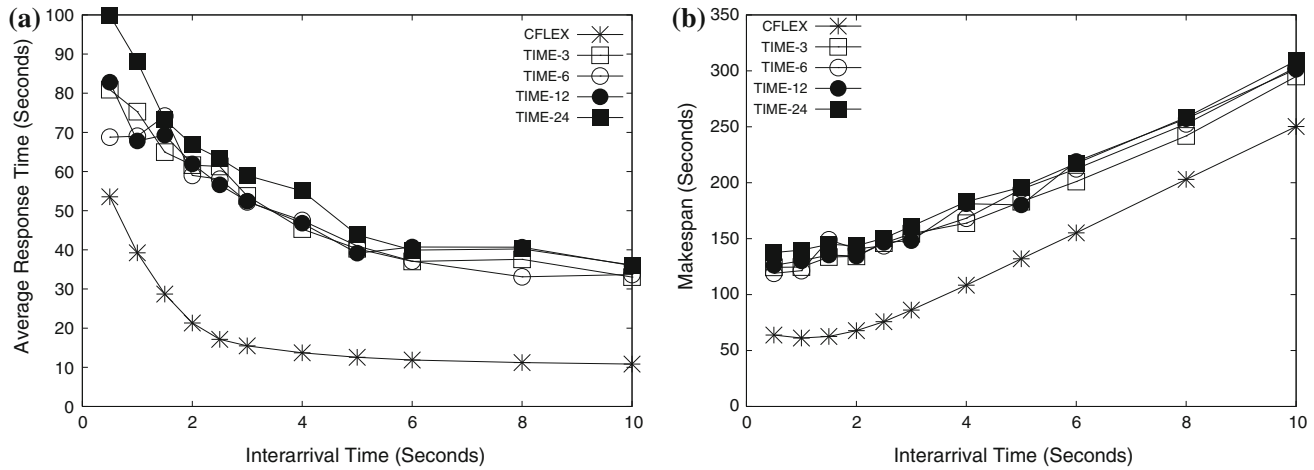

Fig. 18 CIRCUMFLEX versus time-based batching. **a** Average response time, **b** makespan

Figure 17a, b compare CIRCUMFLEX experiments with size-based batching, using batch size N of 1, 2, 3, 4, 6, and 12 queries. Similarly, Fig. 18a, b show comparisons with time-based batching. The batch window W are 3, 6, 12, and 24 s. CIRCUMFLEX remains dominant.

In Fig. 18a, notice that batching with smaller time windows performs consistently better than with larger time windows, for all query interarrival times. This may sound counterintuitive, but recall that the batching heuristic ensures that only one batch runs at a time. As a consequence, the average time between batches spawned is larger than the batch time window. This results in better performance than expected for small batch time windows, especially for small interarrival times.

7 Conclusions

In this paper, we have introduced CIRCUMFLEX, a new scheduler for Map phase jobs in MapReduce environments. This scheme has major advantages over the previous AKO scheme. CIRCUMFLEX is a two stage approach. In the first stage, *cyclic piggybacking* provides a natural and effective technique for amortizing the costs of shared scans. Jobs are decomposed into a number of subjobs, which are related by natural precedence constraints. In the second stage, the resulting precedence scheduling problem is solved for any of a variety of metrics, including average response time, average stretch, and maximum stretch.

Of course, CIRCUMFLEX works best in an environment in which many closely arriving jobs scan the same dataset or datasets. Nevertheless, the scheme will work entirely satisfactorily even if all jobs scan separate datasets. In particular, such a scenario will not cause any significant additional overheads relative to the original MapReduce scheduling paradigm and still provide some performance gains. If employed in the environment for which it was designed, the benefits can be large. For instance, we observed great performance improvements using cyclic piggybacking in our Clydesdale prototype, both in terms of makespan (up to a factor of 3) and average response time (up to a factor up to 5).

Our experiments comparing CIRCUMFLEX with FLEX and BATCH show that the benefits of CIRCUMFLEX can be large. Moreover, CIRCUMFLEX works well in a general overlapping dataset environment, resulting in a number of subjobs related by more arbitrary precedence constraints. In this scenario, CIRCUMFLEX can optimize any minimax metric, including maximum stretch.

In the future, we plan to implement and evaluate using Clydesdale a generalized version of CIRCUMFLEX capable of dealing with overlapping datasets. We are currently working on implementing a split elimination feature in Clydesdale which will make the overlapping datasets scenario commonplace. Using this feature, Clydesdale will be able to skip scanning a split if it can guarantee, based on some precomputed statistics, that none of the records stored in this split can satisfy the query. Thus, two different queries on the same dataset will in reality scan different subsets of the dataset's splits, resulting in an overlapping dataset problem. We also plan to work on efficient algorithmic solutions for truly large CIRCUMFLEX problem instances.

References

1. Agrawal, P., Kifer, D., Olston, C.: Scheduling shared scans of large data files. Proc. VLDB Endlow. 958–969 (2008)
2. Apache ZooKeeper: http://hadoop.apache.org/zookeeper
3. Balmin, A., Kaldewey, T., Tata, S.: Clydesdale: Structured data processing on Hadoop. Proceedings of SIGMOD (2012)
4. Blazewicz, J., Ecker, K., Schmidt, G., Weglarz, J.: Scheduling in Computer and Manufacturing Systems. Springer, New York (1993)
5. Blazewicz, J., Kovalyov, M., Machowiak, M., Trystram, D., Weglarz, J.: Malleable task scheduling to minimize the makespan. Ann. Oper. Res. **129**, 65–80 (2004)
6. Candea, G., Polyzotis, N., Vingralek, R.: Predictable performance and high query concurrency for data. VLDB J. **20**(2), 227–248 (2011)
7. Coffman, E., Garey, M., Johnson, D., Tarjan, R.: Performance bounds for level-oriented two-dimensional packing problems. SIAM J. Comput. **9**(4), 808–826 (1980)
8. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. ACM Trans. Comput. Syst. **51**(1), 107–113 (2008)
9. Floratou, A., Patel, J., Shekita, E., Tata, S.: Column-oriented storage techniques for MapReduce. Proc. VLDB Endow. **4**(7), 419–429 (2011)
10. Hadoop. http://hadoop.apache.org
11. Harizopoulos, S., Ailamaki, A., Shkapenyuk, V.: QPipe: a simultaneously pipelined relational query. Proceedings of SIGMOD (2005)
12. Hunt, P., Konar, M., Junqueira, F., Reed, B.: ZooKeeper: wait-free coordination for internet-scale systems. Proceedings of USENIX (2010)
13. Ibaraki, T., Katoh, N.: Resource Allocation Problems. MIT Press, Cambridge, MA (1988)
14. Kaldeway, T., Shekita, E., Tata S. Clydesdale: Structured data processing on MapReduce. Proceedings of Extending Database Technology (2012)
15. Knuth, D.: The Art of Computer Programming. Addison-Wesley, Reading, MA (1998)
16. Leung, J., E.: Handbook of Scheduling. Chapman and Hall/CRC, London (2004)
17. Nykiel, T., Potamias, M., Mishra, C., Kollios, G., Koudas, N.: MRShare: sharing across multiple queries in MapReduce. Proc. VLDB Endow. **3**(1–2), 494–505 (2010)
18. O'Neil, P., O'Neil, E., Chen, X.: The Star Schema Benchmark (SSB). http://www.cs.umb.edu/poneil/StarSchemaB.PDF
19. Pinedo, M.: Scheduling: Theory, Algorithms and Systems. Prentice Hall, Englewood Cliffs, NJ (1995)
20. Qiao, L., Raman, V., Reiss, F., Haas, P., Lohman, G.: Main-memory scan sharing for multi-core CPUS. Proc. VLDB Endow. 610–621 (2008)
21. Schwiegelshohn, U., Ludwig, W., Wolf, J., Turek, J., Yu, P.: Smart SMART bounds for weighted response time scheduling. SIAM J. Comput. **28**, 237–253 (1999)
22. Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., Murthy, R.: Hive—a petabyte scale data warehouse using Hadoop. International Conference on Data Engineering (2010)
23. Turek, J., Wolf, J., Yu, P.: Approximate algorithms for scheduling parallelizable tasks. Proceedings of SPAA (1992)

24. Vernica, R., Balmin, A., Beyer, K., Ercegovac, V.: Adaptive MapReduce using situation-aware mappers. Proceedings of Extending Database Technology (2012)
25. Wang, X., Sarma, A., Olston, C.,Burns, R.: CoScan: Cooperative scan sharing in the cloud. Proceedings of SOCC (2011)
26. Wolf, J.L., Rajan, D., Hildrum, K., Khandekar, R., Kumar, V., Parekh, S., Wu, K.-L., Balmin, A.: FLEX: a slot allocation scheduling optimizer for MapReduce workloads. Proceedings of Middleware (2010)
27. Wolf, J.L., Squillante, M.S., Turek, J.J., Yu, P.S., Sethuraman, J.: Scheduling algorithms for the broadcast delivery of digital products. IEEE Trans. Knowl. Data Eng. **13**(5), 721–741 (2001)
28. Zaharia, M.: Hadoop fair scheduler design document. http://svn.apache.org/repos/asf/hadoop/mapreduce/trunk/src/contrib/fairscheduler/designdoc/fair_scheduler_design_doc.pdf
29. Zaharia, M., Borthakur, D., Sarma, J., Elmeleegy, K., Shenker, S., Stoica, I.: Job scheduling for multi-user MapReduce clusters. Technical Report EECS-2009-55, UC Berkeley Technical Report (2009)
30. Zaharia, M., Borthakur, D., Sarma, J., Elmeleegy, K., Shenker, S., Stoica, I.: Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. Proceedings of EuroSys (2010)
31. Zukowski, M., Héman, S., Nes, N., Boncz, P.: Cooperative scans: dynamic bandwidth sharing in a DBMS. Proc. VLDB Endow. 723–734 (2007)