# SpongeFiles: Mitigating Data Skew in MapReduce Using Distributed Memory

Khaled Elmeleegy*
Turn Inc.
kelmeleegy@turn.com

Christopher Olston*
Google Inc.
olston@google.com

Benjamin Reed*
Facebook Inc.
br33d@fb.com

## ABSTRACT

Data skew is a major problem for data processing platforms like MapReduce. Skew causes worker tasks to spill to disk what they cannot fit in memory, which slows down the task and the overall job. Moreover, performance of other jobs sharing same disk degrades. In many cases, this situation occurs even as the cluster has plenty of spare memory—it is just not used evenly.

We introduce SpongeFiles, a novel distributed-memory abstraction tailored to data processing environments like MapReduce. A SpongeFile is a logical byte array, comprised of large chunks that can be stored in a variety of locations in the cluster. Spilled data goes to SpongeFiles, which route it to the nearest location with sufficient capacity (local memory, remote memory, local disk, or remote disk as a last resort). By enabling memory-sapped nodes to tap into the spare capacity of their neighbors, SpongeFiles minimize expensive disk spilling, thereby improving performance. In our experiments with Hadoop[1] and Pig[2], SpongeFiles reduce overall job runtimes by up to 55% and by up to 85% under disk contention.

## Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

## 1. INTRODUCTION

MapReduce environments are the primary platform for processing web and social networking datasets (e.g. at Google, Yahoo, and Facebook). Such data tends to be heavily skewed (e.g. millions of anchortext snippets referring to a single web site), and are also subject to "spamming" efforts by automated agents ("bots") that can exacerbate skew issues. Machine learning techniques are relied on heavily to make sense of the data, and sometimes high uncertainty causes the default return value of a classifier or extraction function (e.g. "unknown topic" or "unknown city") to be assigned to a large fraction of the data items.

Figures 1(a) and 1(b) show the data skew experienced in a production multi-thousand-node Hadoop cluster at Yahoo! over a one-month period. Figure 1(a) shows the reduce task input size distribution across all jobs. It also shows the distribution of the average input size per reduce task per job. The maximum input size is about eight orders of magnitude larger than the median. To study intra-job skew, Figure 1(b) shows the distribution of the unbiased estimator of the *skewness*[3] [4] of the same-job reduce task input sizes. Skewness below -1 or above +1 is considered highly skewed, which is the case for a big fraction of the jobs studied.

Data skew has long been a thorn in the side of shared-nothing data processing systems [10, 29], including MapReduce-based ones [9, 12]. It can cause the data assigned to one processing node to overwhelm that node's memory capacity. The standard way to handle this in database management systems is to have the application *spill* the data it cannot fit in memory into disk [23, 26, 28]. Similarly, as shown in Figure 2(a), MapReduce-based systems have their tasks spill their data to disk if their memories are overwhelmed. At a high level, application-level data spilling is analogous to demand paging in virtual memory. There are key differences though. Unlike the kernel, the application has full knowledge of its data access pattern as it is aware of its execution plan. Hence, it can make perfect memory replacement decisions, which is not the case when using standard cache replacement policies in the kernel's virtual memory system. That is a key reason application-level spilling is the method of choice in data management systems.

Spilling to disk causes a major slowdown. The running time of a parallel job is tied to the makespan, so a disk spill on one node can slow the entire job down substantially.[4] Moreover, it disrupts the sequential access pattern of the disk characteristic to MapReduce workloads. As while a task is sequentially reading its input from disk, it may be spilling to the same disk.

*Majority of this work was done at Yahoo!Research–the authors' previous employer.

[1]Hadoop is an open source implementation of MapReduce.

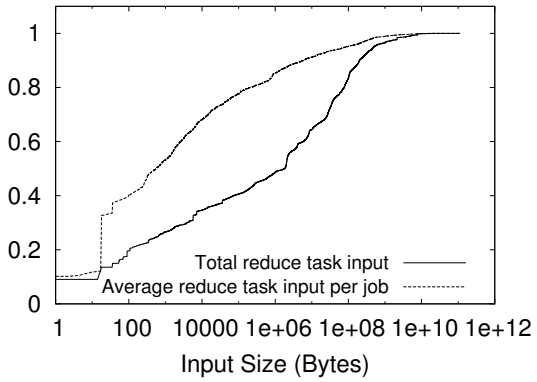
[2]A Pig query is translated to one or more MapReduce jobs.

[3]Skewness is a measure of the asymmetry of a probability distribution. Qualitatively, a negative skew indicates that the tail of the distribution is to the left of its mean. Conversely, positive values indicate that the tail is to the right of the mean.

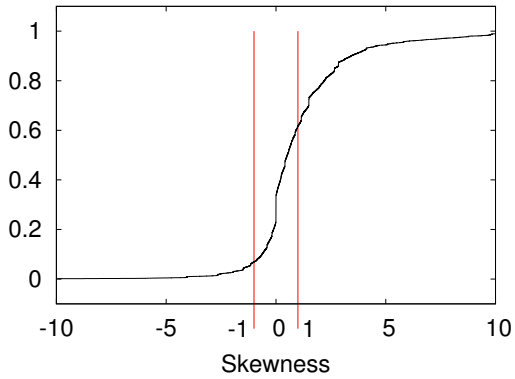
[4]In the MapReduce literature, slow tasks slowing down an entire job are called *stragglers* or *outliers* [1, 9, 19, 32]. However, only straggling due to faulty or slow nodes, not data skew, was covered.


*SIGMOD'14,* June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2595634.




(a) The first curve shows the distribution of inputs across all reduce tasks, while the second curve shows the distribution for the average input per reduce task per job. Please note that the X-axis is in log scale.




(b) Skewness of reduce tasks' input sizes within the same job. Note that a skewness value below -1 or above 1 suggests that the distribution is highly skewed.

**Figure 1: CDFs depicting data skew in a production Yahoo! cluster over a duration of one month.**

Complete spill avoidance is often impractical and sometimes is even impossible. Avoiding spilling requires either providing every task with enough memory for its working set or using data skew avoidance techniques to eliminate data skew. First, always providing sufficient memory is often difficult to do efficiently and sometimes is even impossible. This is because tasks' memory requirements are only known at runtime as they execute arbitrary user code having arbitrary inputs. Further, the required memory may not exist at the node executing the task, especially that, as seen in Figure 1(a), many tasks have inputs larger than any single node's total physical memory. However, although wasteful, over-provisioning memory allocated to tasks can mitigate spilling. Second, data skew avoidance techniques are often ineffective or impractical as we will explain in Section 2.2.

Unfortunately due to data skew, while some tasks experience spilling to disk, the rest of the cluster has plenty of idle memory. A lot of previous work have studied utilizing remote idle memory, when a machine is under memory pressure [6,14]. This work was primarily done in the context of the kernel's virtual memory system, where local memory pages are paged to remote memory. This has the shortcoming of paging in a single page (typically few KBs) at a time as it is difficult to speculate in the general case, which pages will be accessed in the future. Consequently, each remote page access incurs a network round trip, significantly increasing the access time. Further, this previous work was not done at the cloud scale, as tracking individual memory pages on clusters having thousands of machines, each having tens of GBs of memory is both complex and expensive.

Like remote paging, SpongeFiles utilize remote idle memory for spilling, as shown in Figure 2(b), for better performance. However, unlike remote paging, SpongeFiles operate at the application level, giving applications full control over the memory replacement policy. A spilled object is stored in a single SpongeFile, where there is an one-to-one mapping between spills and SpongeFiles. Since applications try to spill the bigger objects to free more memory, a single spill, especially for MapReduce workloads, can be arbitrarily large (usually tens or hundreds of MBs). To accommodate such spills, a SpongeFile is composed of large chunks (measured in MBs). A chunk is allocated from local memory or remote memory depending on free space availability, with disk chunks used as a last resort. Finally, Sponge- Files export a file interface to be backward compatible with different spilling applications.

Unlike remote paging, SpongeFiles are not as vulnerable to remote memory latency and scalability problems with respect to supporting large clusters with huge memory pools. A SpongeFile is either read or written sequentially in its entirety. Hence, each remote chunk is accessed sequentially and in full, amortizing the network latency over the whole chunk's transfer time. Moreover, chunks within the same file are accessed sequentially as well, which allows for perfect prefetching of chunks to further mask the network latency. Furthermore, because memory chunks are typically large, cluster-wide memory tracking can scale to large clusters with substantial memory pools.

Note that in the rest of this paper, we limit the discussion of data skew to the context of Hadoop's MapReduce and Pig [3]. However, data skew can happen in other cloud data processing platforms like BigTable [5], HBase [2], and PNUTS [7], where some nodes in the cluster may end up with larger active working sets than others. Hence, all these other platforms can use SpongeFiles to mitigate data skew.

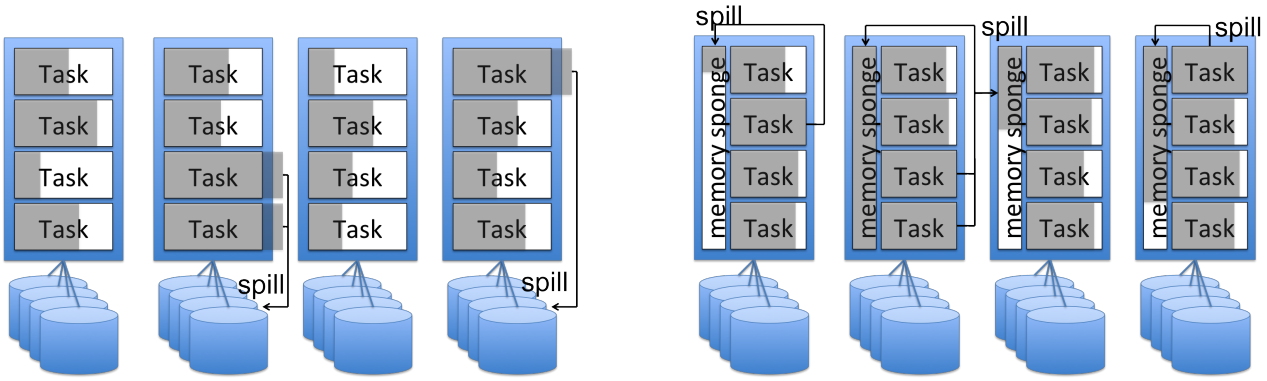
The rest of the paper is organized as follows. Section 2 provides the necessary background. Section 3 describes our design and implementation of the SpongeFile. Our evaluation of the performance of SpongeFiles using realistic workloads is presented in Section 4. Then, related work is discussed in Section 5. Finally, Section 6 concludes the paper.

## 2. BACKGROUND

This section gives background about two topics. First, it talks about data spilling in the Pig/Hadoop data processing stack, providing some necessary background about the execution model. Then, it discusses data skew avoidance methods and their limitations in practice.

### 2.1 Data Spilling in Pig/Hadoop

Pig takes high-level structured dataflow scripts or queries written in Pig Latin [24], translates them into MapReduce




(a) Current spilling approach. (b) Sponge Spilling.

**Figure 2: Memory usage and spilling in Hadoop**

execution plans, and runs the plans in Hadoop. Pig and Hadoop are open-source, run on tens of thousands of computers at large internet companies, and are available on Amazon's Elastic MapReduce cloud computing service.

### 2.1.1 *Hadoop's Architecture*

The Hadoop MapReduce environment runs on an $n+1$ node cluster, consisting of one master node, and $n$ worker nodes. Each worker node communicates with the master to obtain *tasks* to run; a task is a parallel slice of a map or reduce phase of a MapReduce job. In the Pig context, a Hadoop task corresponds to one parallel partition of a query processing pipeline (e.g. project-filter); operators that require hashing or sorting (e.g. most group-by and join algorithms) span the end of a map phase and the beginning of the subsequent reduce phase (see [15] for details).

When the worker node executes a map or reduce task, it runs the task in its own JVM to ensure fault isolation and security in the presence of custom user code. A side effect of this design is that each task has a dedicated memory pool[5], and "leftover" memory of tasks that underutilize their memory pool is wasted. It is possible to tailor memory allocation on a per-job basis, but unfortunately it can be difficult to predict job memory requirements a-priori, and skew can cause highly nonuniform memory requirements across tasks of a given job's map or reduce phase.

Each worker node may execute at most $T$ simultaneous tasks ($T$ is a system-wide parameter). It is useful to think of each worker node as having $T$ task "slots," as depicted in Figure 2. When occupied, each slot contains an executing JVM that consumes certain disk, CPU and memory resources. Ideally, the cluster is configured such that the number of CPU cores, disk heads and slots are well matched to achieve high utilization and good throughput under the anticipated workload. Jobs cap their JVM memory size to $M/T$, where $M$ denotes the total physical memory available for tasks on a node. In other words, tasks are not permitted to swap their virtual memory, because experience has shown that doing so leads to bad random IO access patterns and terrible performance.

[5]This is because Java mandates specifying the required heap at startup time.

### 2.1.2 *Hadoop Spilling*

Hadoop jobs can spill to disk both in the map and the reduce phases.

In the map phase, each map task maps its input into <*key,value*> pairs. These pairs are sorted by the key. To perform this sort, Hadoop uses an in-memory sort buffer, with a default fixed size of 128 MB. If the buffer is filled before all the input is processed, its contents are spilled to local disk. Before the map task finishes, it merges all the spills into a single output file. Note that the input to a map task is typically a single Hadoop Distributed File System (HDFS) block, which has the default maximum size of 128 MB. Hence, for a reasonably provisioned map task, spilling is uncommon.

Each reduce task is assigned a range of the map output key space. The reduce task first collects all its inputs from the outputs of all the map tasks in its job. It then performs a k-way merge (where k is the number of on these sorted inputs) to produce a sorted list of keys, with a list of values associated with each key. This list is then passed to the task's reduce function. The input data can have an arbitrary size as it depends on the number of map tasks in this job and the size of each partition a map task produces for this reduce. For performance reasons, the reduce task tries to perform the merge step in memory. To this end, it assigns a constant fraction of its memory for the merge—by default 70%. If the aggregate size of the input data is more than that, inputs are spilled to disk and they are merged off disk. If the number of input files exceeds a constant value, 10 by default, the merge is done in multiple rounds to reduce disk seeks due to the concurrent reading of a lot of files. Finally, only a fraction of the merged inputs is allowed to remain in memory for consumption by the reduce operation. This fraction depends on the configurable fraction of memory allocated to the reduce task to retain the merged inputs. The default value of this memory fraction is zero. This is to preserve the maximum amount of memory for the higher level application code (e.g. Pig) running in the reduce function. Hence, the default behavior is that the merged inputs are spilled again to disk after the merger.

### 2.1.3 Pig Memory Manager and Spilling

If a task surpasses the JVM memory limit, Pig's memory manager spills data to disk. The memory manager is described in detail in [15]; we give an overview here:

Pig's primary structure for intermediate data is called a *data bag*. A data bag is a collection of tuples that permits insertion and iteration. Bags are registered with Pig's data manager, which keeps track of all bags in the system, estimates their sizes, and keeps tabs on memory utilization relative to the JVM memory size. If available memory becomes low, the JVM delivers an upcall to Pig. In response, Pig invokes its memory manager to spill (portions of) large bags to disk. If spilling is required, a bag is divided into large chunks of size $C$ (typically $C = 10$MB), and chunks are spilled independently.

## 2.2 The Limits of Skew Avoidance

In the database literature, approaches exist to avoid skew and reduce the likelihood of spilling:

- *Skew-resistant partitioning* schemes, most of which focus on joins [11, 16], use data distribution estimates to route data to processing nodes in a balanced fashion. A skew-resistant join algorithm [11] was recently added to Pig, alleviating some, but not all, of the skew-related query[6] slowdown problems.
- *Skew detection and work migration* techniques [1, 18] detect over-stressed nodes and transfer work to other nodes that have more favorable memory, processing or networking conditions.

To utilize these approaches, a Hadoop user has to be experienced with them and employ them in his job. Otherwise, he can use a specialized higher level query language like Pig. These query language provide primitive functions like SUM, AVERAGE,.. with built-in skew-avoidance techniques. However, if the user needs more specialized functions, he is required to write his own. Similarly, users not comfortable with complex expressions composed of these primitive functions, opt for writing their own functions. If these User Defined Functions (UDFs) are not carefully written to properly utilize the query language's API, they become vulnerable to data skew as they contain arbitrary user code.

Data processing platforms like MapReduce are built to enable non-expert programmers to analyze large datasets. Hence, in practice, it is common for users to write queries vulnerable to data skew. The burden is on the system to make these queries perform best.

Moreover, there are cases, where skew-avoidance fundamentally does not work (e.g. in holistic aggregation functions like *median*).

Furthermore, even if data and queries are structured to permit optimal use of memory, the query processing engine itself may not handle all cases optimally: it may materialize some intermediate data collections that could, in principle, be streamed instead. Here again, spilling can occur. In the case of Pig, which has existed as a production system for more than three years and has gone through several overhauls aimed at reducing spilling, spills remain one of the most acute performance issues. Rather than attempting to exhaustively root out all causes of spilling, our approach is to ensure that spilling does not cause a breakdown in performance.

To conclude, skew-avoidance techniques have significant limitations in practice, which explains the magnitude of data skew observed in Figure 1

[6]A Pig query is translated to one or more MapReduce jobs.

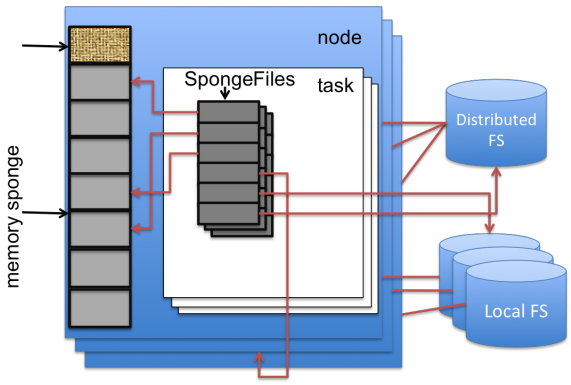



**Figure 3: The structure of SpongeFiles.**

# 3. THE SPONGEFILE

## 3.1 Design

Unlike regular files, a SpongeFile is not meant for persistent storage or data sharing across processes. Instead, it is used to complement a process's memory pool. This difference leads to different design goals and also creates optimization opportunities. The main design goal for SpongeFiles is fast read and write access for bulk data. On the other hand, SpongeFiles are much simpler than regular files. They support very limited number of operations: *create*, *read*, *write*, and *delete*. A SpongeFile has a single writer and a single reader with no concurrent access. It is written once; then it is closed and at a later point in time is read back; finally it is deleted. Consequently, its access pattern is always sequential. Moreover, its lifetime is well known. Furthermore, it does not have durability requirements as it does not persist after it is read. If a SpongeFile's chunk is lost due to a failure before it is read, the task owning the SpongeFile fails. In this case, the Hadoop framework restarts this failed task. Finally, SpongeFiles do not need global name space and consequently do not need a naming service.

A SpongeFile is structured as a list of chunks as shown in Figure 3. Each chunk can lie in the machine's local memory, a remote machine's memory, a local file system, or a distributed file system. Each process (task) is responsible for allocating, maintaining, and freeing chunks for its SpongeFiles.

### 3.1.1 Chunk Allocation

When a SpongeFile is written to, it caches the data in an internal buffer. When the buffer is filled, the SpongeFile attempts to allocate a new free chunk from the local machine's shared memory pool. If the local pool is full, remote memory pools are then tried, with preference given to remote machines whose memory pools are currently being used by the spilling task. This affinity is enforced to improve fault tolerance, because it reduces the number of machines

used by the spilling task. Hence, it reduces the probability of failure of any of the task's components, reducing the task's overall failure probability. In-memory chunks have fixed size to simplify memory allocation. The SpongeFile's internal buffer size is set to have the the same size of the in-memory chunk size. This size should be chosen to be fairly large (in the order of megabytes) to amortize the setup cost for storing the chunk, e.g. the network round trips, when storing the chunk to remote memory. Since SpillFiles are used to spill large bulks of data, using large chunk sizes will not result in significant internal fragmentation.

Depending on the location of the allocated chunk, the SpongeFile uses the corresponding allocator. For in-memory chunks, SpongeFiles have two different allocators—one for local memory and another for remote memory. For on-disk chunks, SpongeFiles rely on the underlying file systems for space allocation.

### *Local Memory Chunk Allocator*

Each machine reserves a separate memory pool *(memory sponge)* that is shared between tasks running on the same machine. In Hadoop, all the tasks have this memory pool outside their JVMs' heaps. This pool is mapped to each JVM's address space using memory mapped files. As shown in Figure 3, the pool is divided into multiple fixed, equal-sized chunks plus a region for the pool's meta data. The meta data includes a global lock to synchronize access to the metadata by different tasks, plus an entry per chunk indicating which task in the cluster is using it. Each entry includes the process ID and the IP address of the machine, where it runs. Free chunks have a special value in their corresponding metadata entries representing that they are free.

When a SpongeFile needs to allocate a new free chunk from the shared pool, it first acquires the pool's lock using a spin lock. It then tries to find a free chunk. If a chunk is found, its entry is updated to reflect the new owner task; then the lock is released and a handle to the chunk is returned. The handle is used to update the SpongeFile's private metadata[7] to point to the allocated chunk in the shared pool. This is used when the SpongeFile is read to index into the corresponding location in memory sponge. Otherwise if no free chunk is available, an error is returned after the lock is released.

Note that sponge memory's metadata is not accessed (avoiding potentially expensive synchronization) when reading from or writing to a SpongeFile except for chunk allocation, when the file is grown.

### *Remote Memory Chunk Allocator*

For a SpongeFile to get access to remote memories in the cluster, all machines run *sponge servers*. A sponge server shares the local sponge memory with the local tasks using the local allocator explained above. It also exports the amount of free space in its local sponge to the cluster. Moreover, it receives and serves allocation requests from remote SpongeFiles.

When storing a chunk in remote memory, the SpongeFile first finds a server with free space. It then writes its data to the newly allocated chunk and gets back a handle to this chunk, which it stores in its metadata.

For optimal remote allocation decisions, a SpongeFile needs current, global, and consistent view of the remote free space. To achieve this, it would require significant communication and synchronization overhead as this state is distributed across many nodes and is updated frequently. Instead, we use a simpler approach with relaxed consistency. In our approach, we rely on a single memory tracking server[8] that periodically (e.g. every one second) polls all the sponge servers for free space. It then constructs a list of sponge servers with free memory. When a SpongeFile is created, it queries the memory tracking server to get the list of sponge servers with free memory. If it needs to allocate a remote memory chunk, it tries a sponge server from the list. Since the information received from the memory tracker could be stale, the sponge server may no longer have free memory. Consequently, the SpongeFile tries the rest of the servers in the free list one at a time until it finds a server with free memory to store its chunk. If no free remote memory is found, the SpongeFile falls back to allocating an on-disk chunk. Note that while some of the information in the free list of sponge servers could be stale, if the overall sponge memory is reasonably provisioned such that sponge memory utilization in the cluster is not very high, the remote allocator will likely succeed. Conversely, under high memory utilization, the remote allocator may fail even though there is some free remote sponge memory. In this case, the SpongeFile will fall back to the classical technique of spilling to disk. We believe that this is the right tradeoff as we are trading full utilization of sponge memory under heavy load for lighter weight remote memory allocation. Arguably, under high sponge memory utilization, not being able to allocate all the free sponge memory is useful. This is because unused memory is used by the operating system's buffer cache to improve disk's performance, which might be stressed at this point.

Finally, in many data centers, including the Yahoo! ones, cross-rack bandwidth becomes a bottleneck at times due to oversubscribed off-rack network links, so spilling across racks may not be advisable. Hence, in our design, we restrict remote spilling to nodes on the same rack, where network bandwidth is plentiful (10 Gb links are now common place). Given that tasks with huge datasets are uncommon (see Figure 1(a)) and given that machines on a single rack often have plenty of memory, skew can be absorbed within a rack. For example, at Yahoo!, each rack has 40 machines, each having 16 GB of memory running 10 tasks per machine, while the biggest input size for a task in Figure 1(a) is 105 GB.

Note that, for uniformity, the remote allocator using the sponge server could have been used for both remote and local in-memory chunk allocation. However, as we will see in Section 4.1, the local allocator is much more efficient. Hence, we use it for local chunk allocation.

### *Disk Chunk Allocator*

In the unlikely event of a SpongeFile not finding free memory in the local and remote memory, it falls back to spilling to disk in the local file system. In this case, the chunk is stored as a local file on disk, where the SpongeFile relies on the

[7]One can think of the SpongeFile's metadata as an inode. However, it is private to the SpongeFile and is only maintained by the file itself.

[8]The memory tracking server can run on any node. A leader election protocol using a coordination service like Zookeeper [17] can be used for that. Since the server is stateless, it can restart on any node in the event of failure.

underlying local file system for the chunk allocation. If the local file system does not have free space either, then, as a last resort, the distributed file systems are tried for the chunk allocation.

On-disk chunks need not have fixed sizes, as allocation is done by the underlying file systems, which use fixed size disk blocks anyway. Moreover, for better performance, it is preferable to keep disk chunks as large as possible to reduce the number of files stored on the underlying filesystem. This keeps the data contiguous on disk and reduces expensive file systems metadata operations. Consequently, when a SpongeFile has a new chunk to store, if there is no free in-memory space available, and if the last stored chunk was on-disk, the new chunk is appended to this on-disk chunk generating a single, larger, on-disk chunk.

#### 3.1.2 Optimizing Reads and Writes

Since SpongeFiles are only accessed sequentially, this creates optimization opportunities. For reads, SpongeFiles prefetch the next non-local-memory chunk. This allows for overlapping computation with IO. Similarly, for writes to non-local-memory chunk, SpongeFiles write the chunk asynchronously to the underlying media to overlap IO with computation as well.

#### 3.1.3 Garbage Collection

When a SpongeFile is deleted, it frees all its allocated chunks by invoking the corresponding chunk deallocator for all of its chunks. Tasks using SpongeFiles should delete their SpongeFiles before they exit. However, tasks may fail to delete their SpongeFiles due to failures or bugs. To handle this, sponge servers perform periodic garbage collections. In a garbage collection, the server checks for chunks in the local sponge, which are owned by dead tasks. These orphaned chunks are then freed. The sponge server checks for liveness of local processes itself, while when it needs to check the liveness of a remote process, it consults the corresponding sponge server to do it on its behalf.

Note that sponge servers and the memory tracker are stateless. Hence, they can easily tolerate failures.

For on-disk chunks, Hadoop tasks write their temporary files into local directories named after the task. When a job ends successfully or unsuccessfully, the Hadoop framework cleans up its directory. Hence, on-disk chunks would be cleaned up automatically by the Hadoop framework. If SpongeFiles are used by non-Hadoop applications, the sponge server could perform the same thing like Hadoop by deleting the directories of dead processes, where its on-disk chunks reside.

#### 3.1.4 Access Control

SpongeFiles are designed to be used in a collaborative environment. However, if access control is needed, tasks can encrypt their chunks before storing them, as once stored the data can be accessible by anyone in the cluster. Moreover, if enforcing quotas is needed, enforcement can be done distributedly, where each task can be prevented from allocating more than a certain amount per node. In this case, each sponge server can stop allocating chunks to a task beyond its per-node quota. Sponge servers can also check for locally allocated chunks by tasks exceeding their quota limits. For these offending tasks, some corrective action can be taken, e.g. killing the task and reclaiming its allocated space.

#### 3.1.5 Discussion

Network spilling is fundamentally superior to disk spilling. This because unlike for network, disk's throughput degrades substantially (possibly by many orders of magnitude), when there is concurrent access due to disk seeks. Often, spilling incurs concurrent access to many spill files. For example, a reduce task, during the k-way merge explained in Section 2.1.2, reads many files simultaneously. So even though each of these files is individually accessed sequentially, this merge involves concurrent access to many files (one file per map output), which can be a significant number depending on the job size. This results in overall random disk access.

It is impractical to provision machines with disks having aggregate random IO bandwidth comparable to available network bandwidth as this may involve hundreds of disks per machine. Even if it were possible to provision machines with sufficient random disk IO bandwidth (e.g. using SSDs), it is inefficient to provision each machine for the infrequent peak load. Conversely, network spilling is more efficient as it allows for pooling the cluster's resources.

### 3.2 Implementation

We implemented SpongeFiles as a Java library to be used with Hadoop and the software stacks that run on top of it. We chose the fixed chunk size for in-memory chunks to be one MB to balance between smaller internal fragmentation and the setup cost (network round trips) for writing a chunk over the network. Since Java supports memory mapped files only up to 2 GB, we implemented the sponge memory as multiple segments to be able to scale to larger sponge memory sizes. Each segment is a separate memory mapped file. Allocators try to allocate memory from any of the segments. Finally, in our current SpongeFiles implementation, we do not support access control. We leave this as future work.

We integrated SpongeFiles with two applications—Hadoop and Pig. In both applications, each spilled object is written into a separate SpongeFile. For Hadoop, in the reduce tasks, we modified the merger of the shuffled map outputs from being done over the disk to be done over SpongeFiles. For Pig, its DataBags were modified to self spill to SpongeFiles instead of disk when Pig's memory manager detects memory pressure.

## Limitations

Since Hadoop tasks run arbitrary user code, Hadoop uses JVMs to execute these tasks for isolation and fault containment. In production Hadoop deployments, the common practice is not to oversubscribe the machine's physical memory. Instead, each JVM is restricted to use memory up to a fixed threshold, such that the sum of memories used by all JVMs on a single machine does not exceed the machine's total physical memory size. This avoids thrashing as it is hard for the OS to make informed decisions about which memory pages to swap. For example, it may swap active pages while keeping *garbage* pages in memory. Java[9] provides a configuration parameter that caps the heap size. This parameter is only configurable at start-up time though. Java then can on-demand grow its heap at runtime up to this cap. Although restrictive, this limitation is useful in preventing runaway tasks from exhausting the machine's memory forcing it into thrashing. It also handles the case, when multiple

[9]This is in reference to Java 7 and before.

tasks each having a large dataset accidentally run simultaneously. Instead of having the kernel thrash the memory (in case the heap size is not capped, e.g. if tasks were implemented in c), control is returned to the application to do targeted application-level spilling.

As a work around to the limitation of not being able to grow the JVMs' heap size at runtime beyond the preconfigured threshold, while input sizes can vary wildly across tasks, we propose that machines' memories are structured as follows. All JVMs have their heap sizes capped to a smaller size that is needed to run small tasks. All the rest of the physical memory in the machine is assigned to the shared sponge memory. Tasks with larger working sets and consequently larger memory footprints are forced to spill to sponge memory. Although this is suboptimal due to copying, serialization, and deserialization overheads, the performance hit is relatively insignificant as we will see in Section 4.2.3. On the other hand, this allows for efficient sharing of memory across the cluster improving its overall utilization.

# 4. EVALUATION

We conducted two sets of experiments: (1) Microbenchmarks to measure the raw performance of various spill media (local sponge memory, remote sponge memory, disk with and without contention) and (2) Macrobenchmarks on end-to-end workloads to see what kind of overall performance differences one can expect. Finally, we study the effectiveness of SpongeFiles in practice by examining relevant production measurements.

## 4.1 Microbenchmarks

In this experiment, we spill a 1 MB buffer 10,000 times to disk and memory using different configurations. In each configuration, we measure the average spill time and report it in Table 1. For the disk case, we seek to a random offset before every new buffer is written to take into account the disk seek incurred for every write. Also, writing to random offsets reduces the chance that the IO operation is absorbed by the OS's buffer cache.

The machines used have two 2.5Ghz quad core Xeon CPUs with 16GB of memory with 7200RPM 300G ATA drives. They are connected via 1Gb ethernet. The machines ran Red Hat Enterprise Linux Server release 5.3 with kernel version 2.6.18 and Ext4 file system.

In the disk experiments, we used the following three configurations: (1) the experiment is running alone on the machine writing buffers to disk, (2) the experiment runs along with a background disk load. The background disk load is similar to that used in 4.2.3, which is from two tasks in a running Hadoop job performing a grep over a large file, and (3) the experiment is running along with the background disk workload plus a background process generating memory pressure. The memory pressure simulates a busy Hadoop node—it pins 12 GB of memory, which leaves very little memory for the buffer cache, reducing the ability of the operating system to batch disk operations to reorder them in a sequential fashion. Consequently, this leads to more disk seeks, reducing achieved bandwidth.

In the memory spilling experiments, we used three configurations, two of which used local memory and one used remote memory. For the local memory cases, we spilled the buffer directly to shared memory and over a socket to the local sponge server. For the remote memory case, we spilled the buffer across the network to a remote sponge server.

As we see in Table 1, spilling to shared memory is the least expensive, then comes spilling locally via the local sponge server as it involves more processing and multiple message exchanges and context switches between two processes. Remote spilling across the network follows as it is limited by the network link bandwidth. Disk spilling follows, where its performance degrades as more background IO load is added in the system. We note that disk spilling is two orders of magnitude slower than memory.

## 4.2 Macrobenchmarks

We describe the workloads, computing environment, experiments and results in turn.

### *4.2.1 Data and Queries*

Our macro experiments use real web data. The data available to us is a random sample multi-million URLs and associated metadata (e.g. crawl time, domain, language, spam score, inlinks, anchortext). Many web analyses are performed at the domain level (examples of domains are google.com and stanford.edu), but we were unable to obtain a sample of complete domains. Hence we constructed a dataset that resembles one that contains complete samples of 100 domains, as follows: We sampled 100 domains, and filtered our URL sample to retain URLs from those domains only; then we scaled up the dataset such that the size of the largest domain in the sample matches the actual size of that domain on the full web. The resulting dataset is about 10GB in size.

The goal of our macro experiments is to test Hadoop's performance, when using SpongeFiles for jobs/queries having significant data skew. To this end we selected one MapReduce job and two Pig queries from the web search domain that are vulnerable to skew-induced spilling due to a nested data format, holistic function, and/or naive lack of projection.

The MapReduce job computes the median of one billion numbers. It has a single reduce task that computes the median. Hence, there is no data skew across the tasks of the same job. However, the reduce task has an input size of roughly 10GB, which, according to Figure 1(a), is in the very high end of input sizes. Hence, this job represent inter-job data skew.

The two Pig queries are realistic production queries that deal with web crawl data:

**Frequent Anchortext** (holistic UDF over skewed groups): *group web pages by language (English, French, etc.), and for each language group find the k most frequently-occurring anchortext terms.*

The `TopK` UDF uses a simple one-pass algorithm to compute the approximate top-$k$ most frequent items (this query sets $k = 10$). □

**Spam Quantiles** (holistic UDF with internal state over skewed groups, without projection): *group web pages by domain, and for each domain group find the spam score quantiles.*

The `SpamQuantiles` UDF places tuples in an ordered bag, and then traverses the bag in sorted order to determine the quantiles of the spam score column. This query represents

| *Spill medium:* | *Local shared memory* | *Local memory (local sponge server)* | *Remote memory, over the network* | *Disk* | *Disk with background IO* | *Disk with background IO and memory pressure* |
|---|---|---|---|---|---|---|
| Time (ms) | 1 | 7 | 9 | 25 | 174 | 499 |

**Table 1: Spilling cost of a 1 MB buffer to different media.**

a situation in which the skew problem is exacerbated by a hastily-assembled ad-hoc UDF for spam quantiles that neglects the basic optimization step of projecting the data down to just the needed fields. □

Note that the two UDFs above execute in the reduce phase of the MapReduce job. They make multiple passes over the data. Also, note that the spam quantiles query has a naive sub-optimal execution plan. This represents a common case in our workload as explained in Section 2.2.

Both Pig queries have a single reduce task with a large and highly skewed input size. The runtime of this straggling reduce dominates the overall job runtime.

### 4.2.2 Computing Environment

These experiments were run on a 30 node Hadoop cluster. The machines have the same configuration described in Section 4.1. We use standard Java 6 and Hadoop 0.20.2 and Pig 0.7. For both Hadoop and Pig, we used two versions in our experiments – the stock version and a modified version that spills to SpongeFiles.

The cluster has one master node (running the Hadoop JobTracker and NameNode processes) and 29 worker nodes (running TaskTracker and DataNode processes). The machines are co-located within a single rack, connected to a common switch with 1G ethernet.

Each worker has two map task slots and one reduce task slot . Each slot runs a JVM configured with 1 GB of maximum heap size (this is the slot's private memory). We also used 1 GB of sponge memory per node.

It is worth mentioning that although this cluster used in the experiments is much smaller than the clusters used in production at Yahoo!, we believe that it is adequate to evaluate our system and SpongeFiles' spilling. This is because, in our system, SpongeFiles only spill within a single rack, which has at most 40 machines.

### 4.2.3 Experiments

In this section, we evaluate the effectiveness of SpongeFile spilling in reducing the running time of jobs suffering from data skew. Afterwards, we study internal memory fragmentation in sponge memory's chunks. Then, we evaluate how tasks spilling to disk disrupt the execution of other tasks. Finally, we study the performance of spilling to different memory configurations and compare this to the optimal case of no spilling.

#### *Spilling to SpongeFiles Vs Disk*

In this experiment, we ran the three jobs described in Section 4.2.1 with spilling to disk versus spilling to SpongeFiles. We also varied the amount of physical memory per node in the cluster to be either low memory (4 GB) or high memory (16 GB). This is to study the effects of available memory on disk performance (due to the operating system's buffer cache) and relate that to SpongeFiles' performance. In this

| | Input Bytes | Spilled Bytes | Spilled Chunks |
|---|---|---|---|
| Median | 10 GB | 10.3 GB | 10527 |
| Frequent Anchortext | 2.5 GB | 7.2 GB | 7383 |
| Spam Quantiles | 3 GB | 10.2 GB | 10478 |

**Table 2: Statistics about the straggling reduce task processing the large dataset.**

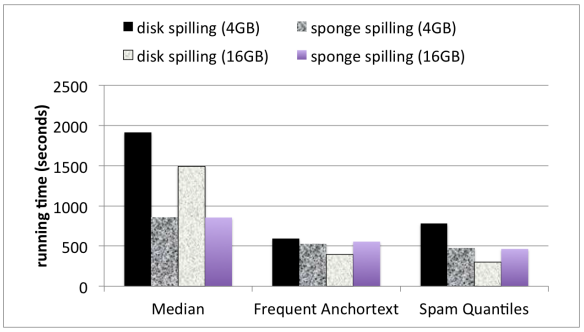



**Figure 4: Comparing the performance of SpongeFiles spilling to that of disk spilling under no disk contention.**

experiment and subsequent experiments, SpongeFiles only use in-memory chunks (either local or remote), as our test cluster had enough memory to absorb spills. This is typically the case in production clusters too as only a small fraction of tasks need spilling, while racks have plenty of memory in aggregate to absorb these spills.

All reported numbers represent averages over three runs, to dampen variance.

Figure 4 shows the running times when jobs are run in isolation, with no other active jobs in the system contending for resources. For each job it shows four configurations, varying two things—spilling to disk versus spilling to SpongeFiles and the amount of physical memory available per node. On the other hand, Table 2 shows some statistics about the three jobs, when using SpongeFiles. Each row shows the input size, the size of the spilled data, and the number of SpongeFiles' chunks spilled for the longest-running reduce task in its corresponding job. We focus on the straggling reduce task because it is the one processing the largest data partition and its runtime dominates the overall job runtime.

In Figure 4, we observe few things. First, when available memory is limited, spilling to SpongeFiles performs better than spilling to disk. Conversely, if memory is abundant performance depends on the amount of data spilled and the time difference between when the data is spilled and when it is read back. For example, in the median job, which is

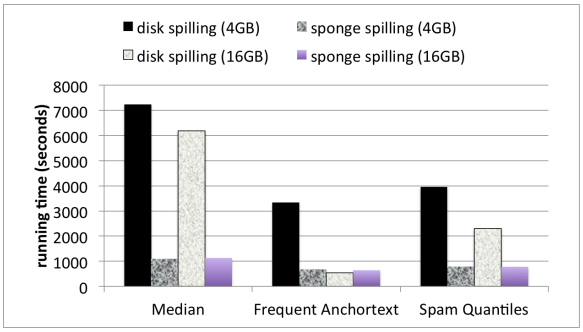



**Figure 5: Comparing the performance of SpongeFiles spilling to that of disk spilling under disk contention.**

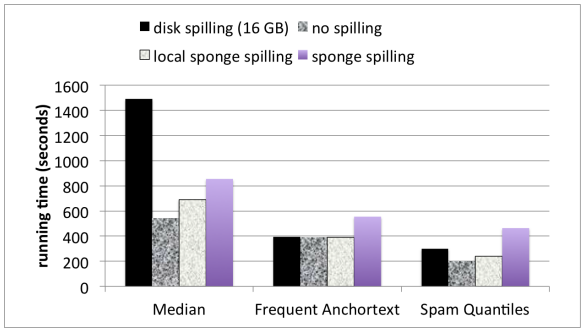



**Figure 6: Comparing the performance of SpongeFiles spilling under four memory configurations—no disk I/O.**

a MapReduce job, the straggling reduce spills all its input data before it starts reading it back to perform the merge process. When spilling to disk, this overwhelms the buffer cache forcing significant disk IO, which hurts performance. Moreover, this spilled input data spans multiple files. As explained in Section 3.1.5, performing a k-way merge of these files introduces disk seeks, which further hurts performance. Consequently, spilling to SpongeFiles provides significantly better performance in this case. On the other hand, for the two other jobs, the input data of the straggling reduce tasks is much smaller. Moreover, when executing the Pig queries, Pig alternates between spilling and reading. Hence, only relatively small amounts of data are spilled before they are read back. This allows the buffer cache to absorb these spills, when memory is abundant. Consequently, spilling to disk performs better in this case as it is effectively spilling to local memory versus spilling to remote memory. Second, unlike disk spilling, we find that SpongeFile spilling does not depend on the amount of physical memory in the system. This is because, it does not rely on the buffer cache, also, because there is enough physical memory in the system such that it does not thrash due to using sponge memory.

### *Effects of Disk Contention*

The above experiment is not realistic enough though as in a typical production environment, there are many jobs running simultaneously. To simulate a real multi-tenant environment in which multiple jobs run concurrently and compete for resources, we repeated the experiment with a background job. For the background job, we used a "grep" MapReduce job that performs a map-only pass over a 1TB of the web dataset, producing disk contention. The background job is submitted to Hadoop following the main job we are measuring, which ensures that the tasks of the background job constantly occupy all map slots not used by the job we are studying.

Figure 5 shows the running times of disk-based spilling and SpongeFile spilling in the presence of background jobs. Again, we notice that the median job performs worst when spilling to disk. This is due to the amount of disk IO it incurs. As we have seen in Table 1, this IO is very expensive due to contention. Using SpongeFiles reduces the job's runtime by over 85% in case of disk contention and memory pressure. Similar behavior is seen for the spam quantiles job. For the frequent anchor text job, spilling to SpongeFiles performs much better than spilling to disk under disk contention and scarce memory. However, when memory is abundant and even with disk contention, spilling to disk performs slightly better than spilling to SpongeFiles. This is because the amount of spilled data in this case is small enough to remain in the buffer cache and not get evicted before it is read again.

### *Fragmentation in In-memory SpongeFile Chunks*

From Table 2, we can compute the fraction of memory wasted in SpongeFile chunks due to internal fragmentation. The wasted memory equals the difference between the number of spilled chunks multiplied by the chunk size (1 MB) and the number of the bytes spilled. We find that it is well below 1%. Hence, one can conclude that for our workloads and for one MB chunks, internal fragmentation is negligible.

### *Effects of Disk Spilling on Other Jobs*

To study the effects of spilling to disk on other tasks in the system, we considered the running times of tasks from the background grep job running concurrently with spilling tasks. We noticed that spilling to disk induces substantial variance in the running times of the background job's tasks: most grep task instances ran for about 16 seconds, whereas "unlucky" ones that overlapped with disk spilling took as much as 39 seconds to complete. This effect hurts system throughput somewhat, but more importantly it reduces performance predictability for all jobs in the system.

### *Performance of Different Memory Configurations*

In this experiment, we investigate the performance of different memory spilling schemes and contrast them with the optimal case, where no spilling takes place. More specifically, for all of the three jobs, we study: (1) spilling to disk, where there is plenty of physical memory (16 GB) that the buffer cache virtually absorbs all the spilling, (2) spilling exclusively to local memory sponge, where a large memory sponge (12 GB) is used to absorb all the spilling, (3) no spilling, where the JVM is given a large heap (12 GB) so that it can fit all the data in its memory without having to spill, and (4) SpongeFile spilling, where each node has 1

GB of memory sponge. Hence, most of the spilling goes to remote memory.

Figure 6 shows the results of this experiment. No spilling performs best for all jobs as data does not go through the overhead of serialization and spilling, then reading and deserialization. Spilling to local sponge memory performs second best for the three jobs. Spilling to disk (buffer cache) performs better than spilling to the SpongeFile for the frequent anchortext and the spam quantile jobs. This because writing to the local memory, for the case of the buffer cache, is faster than writing over the network. For the median job though, spilling to the SpongeFile performs better than spilling to the disk (buffer cache). This is because, as explained in Section 2.1.2, when the reduce task does the k-way merge of its input data, it is done in multiple rounds to decrease the disk seeks that could result from reading multiple files concurrently. Consequently, for the disk case, a lot more data is spilled due to the multiple merges—a total of 16.1 GB Vs 10.3 GB for SpongeFiles spilling. On the other hand, merging off the SpongeFile is done in one round as there are no disk seeks to avoid. Note that for the disk spilling case all the data is in the buffer cache, so no disk seeks would have occurred had all the inputs been merged together in one round. However, the application has no way of knowing whether the data is available in the buffer cache or not. Hence, it assumes the worst, i.e. the data will be read off the disk.

Note that all of the above schemes, except spilling to SpongeFiles, are impractical. This is because they require grossly over-provisioning a machine's memory to accommodate peak load, especially given the amount of data skew shown in Figures 1(a) and 1(b). Conversely, when using SpongeFiles, peak load can be easily accommodated due to sharing of memory across many machines. Moreover, the performance of SpongeFile's spilling is comparable to the optimal (no spilling) case.

## 4.3 SpongeFiles in Practice

**Effectiveness:** For maximum effectiveness, SpongeFiles need to keep their data in memory. In the MapReduce context, SpongeFiles are well suited for hosting intermediate data. Consequently, the aggregate size of intermediate data of running jobs should be comparable to the aggregate memory size in the cluster. By studying the distribution of intermediate data sizes in Yahoo! clusters for one month, we found that, at any point in time, the aggregate intermediate data size is at at most 25% of the total cluster's memory size. This is due to multiple reasons. First, many jobs do significant filtering of data at the map phase with 90% of the inputs filtered on average. Second, as reported by Facebook [31], a large fraction of the workload is for small small jobs with small inputs, and consequently small intermediate data. These jobs are ad-hoc queries issued by users to mine the data.

Also, we find that in practice using remote memory is important. In Figure 1(a), we see that some reduce tasks have inputs larger than 105GB, which does not fit into the memory of a single machine.

**Failure Analysis:** Another potential weakness of SpongeFiles is the fact that when a task's data is spread around onto multiple nodes, failure of any of these nodes would cause the task to fail. To study this, we model the probability of task failure due to machine failure as a Poisson process using two parameters: the running time $t$ of the task and the Mean Time To Failure (MTTF) of a machine. If a task's data is spread onto $N$ machines, the probability that one or more of those machines fails (and thus brings down the task) is:

$$P = 1 - (e^{-N \cdot t/MTTF})$$

Yahoo! has tens of thousands of machines in its clusters. The observed failure rate there is roughly 1% per month [13]. This translates to a MTTF of 100 months. In our experiments reported in Section 4.2, the longest running time of any task was about 120 minutes. Hence, the probability of failure would still remain very low even for tasks spilling to many nodes. With these parameters, the additional risk of failure due to cross-memory spilling to remote machines is not significant. In fact, pre-existing task failure causes dominate the task failure likelihood. These causes include overload of the various components of the Hadoop Distributed File System, unhandled exceptions thrown by UDFs and runaway processes.

With very long-running tasks, the probability of failure due to spilling to remote machines can become substantial. However, this increased vulnerability is offset by the fact that SpongeFiles spilling enables long-running tasks to complete more quickly, thereby reducing the duration of vulnerability.

# 5. RELATED WORK

Our work is complementary to skew-resistant partitioning [11, 16, 27]. Whereas skew avoidance aims to minimize or eliminate the need to spill, our approach makes spilling more efficient in cases where it cannot be avoided easily, e.g. due to user-defined processing elements and nested datasets, as discussed in Section 2.2.

Cooperative caching [8] permits nodes in a cluster to read data cached at peer nodes (versus retrieving it directly from a central server). Whereas cooperative caching deals with shared access to portions of a database or file system, our focus is on how to manage temporary data spilled from a worker node.

Network memory has been used by many systems for object storage like Memcached [22] and RAMCloud [25]. Unlike SpongeFiles, Memcached cannot store arbitrarily large objects as its objects do not span multiple machines. RAMCloud is an in-memory ultra-low-latency key-value store, focusing on small objects.

Remote paging systems [20, 21, 30] page to the memory of other machines over the network rather than going to disk. This approach must be implemented in the operating system, which has a limited view of what is happening in the application layer. Our approach takes advantage of a clear understanding of how the memory is used, e.g. the application knows which data to spill and which data to keep. Also, SpillFiles have perfect knowledge of their access pattern, allowing for optimizations like prefetching.

# 6. CONCLUSION

SpongeFiles allow tasks to utilize the aggregate storage resources of the cluster to mitigate the effects of data skew and reduce load on the disk during periods of high memory usage. They do this by exploiting the non-uniformity of the memory utilization in the cluster. This approach is

complementary to skew avoidance: for cases where skew cannot (easily) be avoided, and helps ensure than when spilling needs to happen, it happens as fast as possible. We showed that by taking advantage of the properties of spilling: non-shared storage, sequentially accessed, and a lifecycle that is well-defined and short, we can design a flexible system that can adapt to the dynamic distributed environments of MapReduce clusters. Our experiments show that spilling to SpongeFiles reduce job runtimes by up to 55% in absence of disk contention and by up to 85% when there is disk contention compared to traditional disk spilling. We expect other distributed applications that must handle skewed data or spikes in load can also benefit from SpongeFiles.

# 7. REFERENCES


[1] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. OSDI*, 2010.

[2] HBase: the Hadoop database. `http:///hbase.apache.org/`.

[3] Pig: Query processing system on Hadoop. `http://pig.apache.org`.

[4] M. G. Bulmer. *Principles of statistics*. Courier Dover Publications, 1979.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Computer Systems*, 26(2), 2008.

[6] D. Comer and J. Griffioen. A new design for distributed systems: The remote memory model. In *Proceedings of the Summer 1990 USENIX Conference*, USENIX '90, 1990.

[7] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[8] M. D. Dahlin, O. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *In Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, 1994.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.

[10] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *CACM*, 35(6), 1992.

[11] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proc. VLDB*, 1992.

[12] D. J. DeWitt and M. Stonebraker. MapReduce: A major step backwards. `http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/`.

[13] K. Elmeleegy. Piranha: Optimizing short jobs in hadoop. *Proc. VLDB Endow.*, 6(11):985–996, Aug. 2013.

[14] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 201–212, New York, NY, USA, 1995. ACM.

[15] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: The Pig experience. In *Proc. VLDB*, 2009.

[16] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proc. VLDB*, 1991.

[17] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIXATC'10: Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 11–11, 2010.

[18] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in mapreduce applications. In *SIGMOD '12: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36, New York, NY, USA, 2012. ACM.

[19] J. Lin. The curse of Zipf and the limits to parallelization: A look at the stragglers problem in MapReduce. In *Proc. Workshop on Large-Scale Distributed Systems for Information Retrieval*, 2009.

[20] E. Markatos, E. P. Markatos, G. Dramitinos, and G. Dramitinos. Implementation of a reliable remote memory pager. In *In USENIX Annual Technical Conference*, 1996.

[21] I. McDonald. Remote paging in a single address space operating system supporting quality of service. Technical report, Department of Computing Science, University of Glasgow, Scotland, UK, 1999.

[22] Memcached: A distributed memory object caching system. `http://memcached.org/`.

[23] Microsoft. Sql server 2008 r2 documentation. `http://msdn.microsoft.com/en-us/library/ms191514.aspx`.

[24] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proc. ACM SIGMOD*, 2008.

[25] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *SOSP '11: Proceedings of the twentieth ACM symposium on Operating systems principles*, 2011.

[26] Oracle. Oracle database concepts – memory architecture, 10g release 2. `http://docs.oracle.com/cd/B19306_01/server.102/b14220/memory.htm`.

[27] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 61–72, New York, NY, USA, 2012. ACM.

[28] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[29] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proc. VLDB*, 1991.

[30] D. Williams, H. Jamjoom, Y.-H. Liu, and H. Weatherspoon. Overdriver: handling memory overload in an oversubscribed cloud. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, 2011.

[31] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[32] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 29–42, 2008.