Rock You like a Hurricane: Taming Skew in Large Scale Analytics

Laurent Bindschaedler
Ecole Polytechnique Fédérale de Lausanne (EPFL)
laurent.bindschaedler@epfl.ch

Jasmina Malicevic
Ecole Polytechnique Fédérale de Lausanne (EPFL)
jasmina.malicevic@epfl.ch

Nicolas Schiper*
Logitech
nschiper@logitech.com

Ashvin Goel
University of Toronto
ashvin@eecg.toronto.edu

Willy Zwaenepoel
Ecole Polytechnique Fédérale de Lausanne (EPFL)
willy.zwaenepoel@epfl.ch

ABSTRACT

Current cluster computing frameworks suffer from load imbalance and limited parallelism due to skewed data distributions, processing times, and machine speeds. We observe that the underlying cause for these issues in current systems is that they partition work statically. Hurricane is a high-performance large-scale data analytics system that successfully tames skew in novel ways. Hurricane performs *adaptive work partitioning* based on load observed by nodes at runtime. Overloaded nodes can *spawn clones of their tasks* at any point during their execution, with each clone processing a subset of the original data. This allows the system to adapt to load imbalance and dynamically adjust task parallelism to gracefully handle skew. We support this design by spreading data across all nodes and allowing nodes to retrieve data in a decentralized way. The result is that Hurricane automatically balances load across tasks, ensuring fast completion times. We evaluate Hurricane's performance on typical analytics workloads and show that it significantly outperforms state-of-the-art systems for both uniform and skewed datasets, because it ensures good CPU and storage utilization in all cases.

CCS CONCEPTS

• **Information systems**; • **Applied computing**; • **Computer systems organization → Architectures**; *Dependable and fault-tolerant systems and networks*; • **Networks**;

*Nicolas Schiper was with EPFL when this work was performed.

KEYWORDS

Hurricane, big data, analytics, cluster computing, skew, high performance, task cloning, adaptive work partitioning, merging, repartitioning, load balancing, storage disaggregation, decentralized storage, bags, chunks, fine-grained partitioning, distributed scheduling, batch sampling, late binding.

ACM Reference Format:

Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. 2018. Rock You like a Hurricane: Taming Skew in Large Scale Analytics. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3190508.3190532

1 INTRODUCTION

Application runtimes in data analytics frameworks are unpredictable and underperforming on certain input datasets and software / hardware configurations. These issues often occur because different tasks within a job take different amounts of time to complete, causing a load imbalance where some machines sit idle while waiting for others to finish, thereby limiting the achievable degree of parallelism. Slower tasks can degrade performance for the entire parallel job, resulting in delayed job completion [12], resource underutilization [24, 48], and even application crashes.

Task runtime variance is caused by skew. Tasks may be assigned different amounts of data due to data skew in the partitioning [29, 40]. Such skew occurs intrinsically in many real-world datasets, making it hard to create well-balanced partitions. For example, a web dataset may have millions of records referring to a website, map-reduce algorithms have popular keys, and social networking and graph datasets have high degree vertices. Tasks may also suffer from compute skew, wherein the execution time depends on the data, regardless of its size. For instance, an algorithm may do more processing on some inputs or selectively filter data [23]. Besides data and compute skew, task runtime can also be affected by machine skew, for example, heterogeneous or faulty machines [10]. A search for "skew" in analytics workloads on stackoverflow [9] yields hundreds of relevant results from programmers experiencing painful problems and unexpected crashes due to improper handling of skew.

This paper describes Hurricane, a high-performance analytics system that achieves fast execution times and high cluster utilization with an adaptive task partitioning scheme. The core idea underlying this scheme is *task cloning*, where an overloaded node can clone its tasks on idle nodes, and have *each clone process a subset of the original input*. This allows Hurricane to adaptively adjust parallelism *within* a task, and dynamically improve load balance across nodes based on observed load, *at any point in time*. This is the key to handling highly skewed workloads: underperforming tasks can be split across multiple nodes *dynamically during their execution* and idle nodes can pick up a part of the task load.

By comparison, state-of-the-art frameworks such as Hadoop [25] and Spark [47] struggle to achieve load balance and good parallelism because they rely on static partitioning of work. Partitions are created based on storage blocks or programmer defined split functions, but the partition sizes and processing requirements often depend on the dataset, and are thus known only at runtime. Once partition bounds are fixed, the degree of parallelism for a stage cannot be dynamically adjusted: it is not possible to split the work or increase parallelism within a partition when it takes a long time to process, immaterial of the reason it takes that long. For this same reason, while traditional straggler mitigation techniques such as speculative execution [18] and tiny tasks [35] can help with slow machines, they do not directly address data or compute skew.

Hurricane supports task cloning by combining two novel techniques: *fine-grained independent access to data* and *programming model support for merging*. These techniques enable writing high-performance, skew-resilient applications with minimal programmer effort.

Fine-grained data access enables workers[1] executing tasks to compute on small partitions of *any* input or intermediate data independently of other workers, allowing fine-grained, on-demand task cloning. To support this design, Hurricane stores *all* input and intermediate data in *data bags*. Each data bag corresponds to the input or output of a task, and data is stored in *fixed-size* blocks, called chunks, within bags. A bag does not belong to a worker; rather all workers executing clones of the same task share the bag.

Hurricane supports combining the partial outputs of cloned tasks using an application-specified *merge* procedure whose output is equivalent to the output of a single uncloned task. Our merging paradigm is more general than the traditional *shuffling and sorting* method for combining outputs from different partitions. It not only alleviates the need to sort, but also allows for data records associated with the same key to be simultaneously processed on multiple nodes, providing more flexibility to balance load across partitions in the presence of key skew. We provide further justification for our approach in Section 6.

Efficient cloning of tasks requires good data placement to avoid increasing storage load on overloaded nodes. For example, if all the data is co-located with the node executing a task, cloning to shed load from that node will require redistributing the data. This can add further strain on that node, potentially leading to a storage bottleneck. Hurricane avoids this problem by spreading the chunks in data bags uniformly randomly across all nodes in the cluster, and allows retrieving efficiently them using a decentralized scheme. This approach achieves high cluster-wide storage utilization and throughput, thus ensuring that cloning does not lead to storage bottlenecks.

[1]A worker is a container executing a task on a node.

We have implemented several typical analytics applications on Hurricane. We evaluate the system on a cluster of 32 machines that are connected by a high-speed network. We show that Hurricane achieves load balance and scales with the number of machines in the cluster, the input data size, and the amount of skew. We observe a slowdown compared to uniform partitions of at most 2.4× in a click counting application in the presence of 64× imbalance between partitions. Hurricane can execute skewed hash joins 18× faster than Spark, while keeping the performance degradation with high skew below 2.3×, and outperforms Spark's GraphX [45] by calculating PageRank on real-world graphs 5-10× faster.

The contributions of this paper are four-fold. We present the first analytics system designed to systematically perform adaptive partitioning of work based on observed load during execution, allowing it to improve application runtime by optimizing parallelism and providing load balance for both compute and storage resources. We demonstrate how to implement such a system through a fine-grained, adaptive partitioning scheme based on a task cloning abstraction. We propose a storage architecture which maximizes storage utilization and throughput while allowing workers to efficiently and independently access data. We demonstrate the performance of our system for several typical analytics workloads.

The rest of the paper describes our approach in detail. Section 2 describes the Hurricane programming model, and then, Section 3 presents the design of our system. Section 4 describes the implementation. Section 5 evaluates the performance of Hurricane on typical analytics workloads. Section 6 discusses and compares the related work in the area, and finally Section 7 provides our conclusions.

2 PROGRAMMING MODEL

Hurricane supports a dataflow application model. To achieve good parallelism and load balance even in the presence of high skew, Hurricane clones tasks based on observed load at any point during their execution. This requires programming model support, specifically the ability for multiple workers to *share* data within a partition, as well as the ability to reconcile multiple partial outputs into a single consistent output.

2.1 Application Model

Hurricane applications are specified as a directed graph of tasks, shown as circles, and data bags. The edges in the graph represent the flow of data between tasks and bags, i.e., the outputs of bags are connected to the inputs of tasks, and the outputs of tasks are connected to the inputs of bags.

Executing the application graph creates an execution graph, where nodes in the cluster execute the various tasks on local workers. A

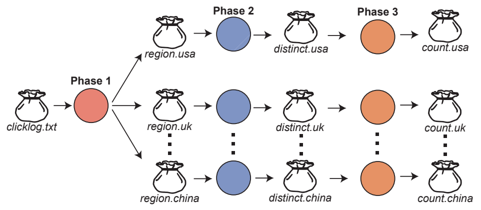


Figure 1: **ClickLog computation graph.**

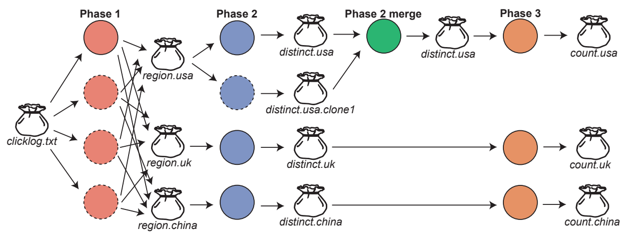


Figure 2: **A possible ClickLog execution graph on 4 nodes. Hurricane automatically cloned the phase 1 task as well as the phase 2 task for the USA region. Note that cloning a phase 2 task requires the introduction of a corresponding merge.**

worker can either execute a task or a clone of a task. The system ensures that tasks only start once their corresponding input bags are ready.

The Hurricane framework may *at any point* decide to clone a task to increase parallelism and ensure faster completion. Cloning tasks is entirely managed by the system. When partial outputs from clones must be reconciled, the application specifies a merge procedure to combine them. If no such procedure is specified, Hurricane simply concatenates the outputs of all clones.

Figure 1 shows the graph topology of a typical Hurricane application called ClickLog that operates on a log of clicks on advertisements to count the number of unique IP addresses from each geographic region. This application uses three types of tasks, whose pseudo-code is shown in Figure 3. Phase 1 tasks map the source bag, containing the click log, into per-region output bags, Phase 2 tasks list the unique IP addresses in each region bag, and Phase 3 tasks count the size of the list.

Figure 2 shows a possible execution graph using 4 nodes. All workers execute tasks or cloned tasks, shown using dashed lines. In this example, the number of clicks on advertisements per region can vary significantly, causing skew in the tasks. As a result, Hurricane may decide to clone some tasks. For example, Phase 1 has one original worker executing the task and 3 clones. A task and its clone run the same code. The Phase 2 task operating on the USA region has two workers associated with it (the original worker and a clone). Phase 2 requires a custom merge, which is executed after all associated workers finish. Note that different tasks (not clones) may also run the same code. For example, Phase 3 has three different tasks, running the same code, but with different input bags.

```
Phase 1 task (input, outputs):
  while ip = input.remove() not empty:
    region = geolocate(ip)
    outputs[region].insert(ip)

Phase 1 merge (partial1, partial2, output):
  output = concat(partial1, partial2) // default merge

Phase 2 task (input, output):
  let distinct be a bitset
  while ip = input.remove() not empty:
    distinct |= ip // set corresponding bit to 1
  output.insert(distinct)

Phase 2 merge (partial1, partial2, output):
  output.insert(partial1 | partial2)

Phase 3 task (input, output):
  output.insert(len(input))

Phase 3 merge (partial1, partial2, output):
  output.insert(partial1 + partial2)
```

Figure 3: ClickLog application code.

2.2 Dynamic Fine-grained Data Sharing

Multiple workers (clones) executing the same task on the same input data require a way to obtain disjoints subsets of the data. Since Hurricane may adjust the number of clones dynamically during task execution, workers should be able to independently and efficiently access finer-grained partitions of the data dynamically at runtime.

Hurricane achieves fine-grained data sharing through a data bag abstraction that workers use to store data and communicate with each other. Each data bag contains fixed-size blocks of data called chunks that are stored in files in our distributed storage service.

Data bags expose two main operations: `Bag.insert(chunk)` and `Bag.remove()`. The first operation inserts a chunk into the bag. The second operation removes a chunk from the bag and returns it to the caller. The bag abstraction guarantees that each chunk in the bag is returned *exactly once*, ensuring that a chunk is processed once (by some task clone). The remove operation fails when a bag is empty, allowing a worker to terminate.

A worker serializes its application-specific data records into a chunk before inserting it into a bag. Similarly, after removing a chunk, it deserializes the chunk into its data records. Hurricane provide a number of typed iterators for serializing and deserializing common formats (integers, floats, strings, tuples, etc.), which can be combined to represent more complex data types (e.g., nested tuples). All serializers ensure that data records do not cross chunk boundaries, thus allowing chunks to be processed independently.

Data bags differ from files in that they allow multiple workers to concurrently insert or remove chunks from the same bag without interference. For instance, in Figure 2, the two workers processing the Phase 2 for the USA region read chunks from the same bag (`region.usa`). This property derives from the use of chunks as the basic indivisible unit of data used by workers. Data bags also support data processing at varying speeds by forcing workers to request individual chunks instead of being assigned key ranges upfront. This *late binding* of data chunks to workers is essential to handle skewed workloads as it makes it possible to dynamically partition the data during task execution.

2.3 Dynamic Merge-based Task Sharing

Multiple workers (clones) executing the same task on different subsets of the same input data may need a way to reconcile their partial outputs into a coherent single output. Ideally, workers should be able to process subsets of the data in isolation, and produce individual outputs that can be *merged* to produce the final output.

Hurricane merges partial outputs through a (possibly null) merge procedure. Some tasks can support multiple workers without any additional merging effort. Examples of such tasks include preprocessing, map tasks (from MapReduce), filters, selects (in SQL), etc. In such cases, it is sufficient to concatenate the chunks produced by each worker into the output bag. In the ClickLog example, this is what happens if multiple workers execute Phase 1. Other tasks however require support for merging the partial results of the concurrent workers. Examples of such tasks include reduce tasks (from MapReduce), counting, sketches [16, 22], groupby, etc. In the ClickLog example, this is the case for tasks in Phases 2 and 3. Often, tasks requiring a merge must produce output satisfying some constraints (e.g., sorted result, aggregation). As a result, an intermediate merge is required to ensure output consistency. Since merging is application-specific, if the task requires a merge, we require the programmer to specify a merge procedure as part of the code for that task.

Specifying a merge procedure amounts to defining a function to combine two partial outputs into one. This merge procedure is relatively easy to write. In most cases, it is of similar complexity as writing a merge combiner in Spark. However, unlike merge combiners, the merge operation is more general. Among other things, non aggregation outputs can be merged, for instance through a merge sort. The merge operation also supports non commutative-associative operators (e.g., unique counts, medians, duplicates removal). For convenience, Hurricane provides a library of typical merge operations.

3 DESIGN

Figure 4 shows the architecture and typical deployment of Hurricane. A Hurricane cluster consists of a set of compute and storage nodes. The cluster administrator provisions nodes for use by Hurricane, either manually or through a resource manager, such as YARN [44]. The compute and storage nodes may be co-located, but are provisioned independently. Then each storage node starts a Hurricane server, and each compute node starts the Hurricane framework and is configured so that it knows the list of storage nodes. The compute nodes run tasks on local workers. Each compute node runs on one or more cores, and workers can be multi-threaded. The storage nodes store all bags spread across the nodes. These bags can be of two types: data bags (DB) and work bags (WB). Work bags are used to schedule tasks.

3.1 Execution Model

Each Hurricane application is associated with an application master that runs on one of the compute nodes. The master drives the

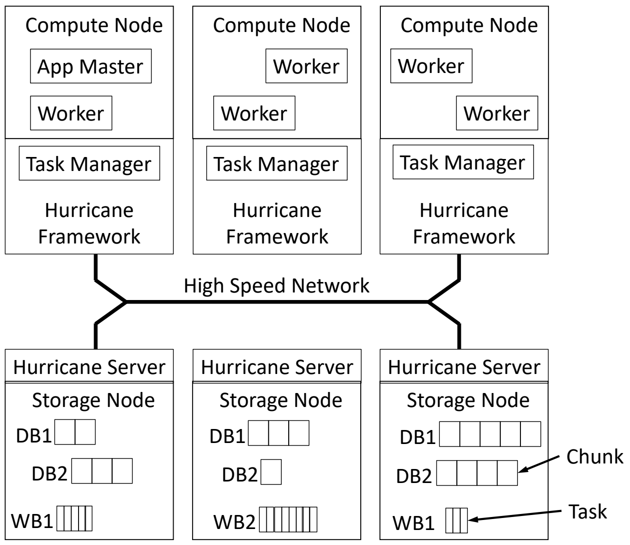


Figure 4: The Hurricane Architecture. Although logically separated, compute nodes and servers can be co-located.

application's computation by invoking functionality in the Hurricane framework. In addition, it monitors application progress, and facilitates the implementation of policies for cloning and resource management. The master is a lightweight component as it relies on distributed work bags to perform most of its functions.

Upon starting, the application master creates a task manager on each compute node that is responsible for executing tasks on local workers. The application master then reads the application graph and schedules tasks for execution. Each task consists of a task blueprint, containing a unique task identifier and the code necessary to execute the task, as well as the identifiers of its input and output bags.

The master maintains the application's progress in the execution graph and schedules new tasks once all the source bags for a task have completed. The overall execution ends once there are no more tasks to be scheduled and no more tasks are being executed.

This execution model ensures that once an input bag becomes empty, it will remain empty, and thus workers know when they are done. For example, in Figure 1, the application master schedules the Phase 1 task, and only when it is finished, schedules all the Phase 2 tasks. Since Phase 3 tasks only depend on the bag containing the distinct list for their respective region, they can be scheduled immediately after the corresponding Phase 2 tasks finish. This simple model suffices for our batch analytics workloads, although we plan to explore more sophisticated dataflow execution models for streaming workloads [32].

3.2 Task Cloning

The application master automatically clones tasks on behalf of the application by modifying the execution graph to add a copy of the cloned task that reads from the same input bag as the original task, as well as (possibly) a merge task. When the task requires a merge, we add a merge task to the execution graph, when the first clone is created. Then, for each clone, the master creates a new bag dependency between the clone and the merge task. Once all the clones complete, we execute the merge task to produce the reconciled output.

Hurricane has two design goals when cloning tasks: automatically adjusting parallelism at runtime with minimal programmer effort, and minimizing the overhead of executing a task on multiple workers. These goals present a trade-off between responsiveness to load imbalance, and the cost of cloning and merging of results.

Dynamic Parallelism. Hurricane clones a task repeatedly until it either runs on every compute node or the system determines that it already benefits from a sufficient degree of parallelism. The application master makes cloning decisions based on two criteria: 1) load information that helps detect task load imbalance, and 2) a cloning heuristic that determines whether cloning will benefit task execution time.

Hurricane detects load imbalance by monitoring two resources throughout the execution of a task: CPU load and network usage. If a worker experiences high CPU load for a prolonged period of time or its network interface is saturated, this is an indication that the worker is experiencing overload, and we should re-evaluate the degree of parallelism in that task. The master then clones the task on an idle compute node if one is available and the cloning heuristic allows cloning, as discussed below.

Hurricane clones a task worker repeatedly until it is no longer overloaded, thus increasing task parallelism only as needed.

Note that Hurricane need not monitor for storage bottlenecks because the storage system is designed to provide the maximum possible bandwidth to applications, and therefore running at peak storage bandwidth is the best case scenario.

Cloning Heuristic. Hurricane only clones a task when it expects that cloning will improve task execution time. Cloning introduces two costs that may require additional computation and IO: 1) loading task state in a new clone, and 2) merging of clone outputs, which introduces an additional dependency in the execution graph. Hurricane estimates these costs to avoid cloning close to task completion.

Consider the ClickLog example from Figure 1. When a Phase 1 worker is overloaded, the application master will always clone the task since it has minimal state and does not require a merge. This process will repeat for each worker in Phase 1 until the task completes or there are no more idle compute nodes. In contrast, when a Phase 2 worker is overloaded, the heuristic may reject cloning if the task is close to completion because the overhead of merging outweighs the benefits. The heuristic may also determine that it is worthwhile cloning, when the task runs for a long time, as in the *region.usa* case. If so, the master performs task cloning by scheduling a copy of the task on an idle node, as it would any other task, and adds the corresponding merge task to the execution graph.

3.3 Storage Architecture

Task clones require efficient and fast access to their subsets/partitions of the input data. This requires careful data placement or else storage can become a bottleneck. For instance, placing all task data on the node executing the task will likely cause that node to experience additional pressure due to data redistribution if its task is cloned.

The bag abstraction, which exposes all data to workers as a distributed storage service, helps avoid storage bottlenecks. Hurricane decouples computation from data, storing data on storage nodes, while processing is performed on separate compute nodes. We leverage this decoupling to achieve near-perfect storage utilization and throughput by uniformly spreading the chunks in data bags across all storage nodes, disregarding any locality concerns. We access these chunks using an efficient and low-latency decentralized scheme, and ensure high storage utilization by prefetching chunks using a batch sampling strategy.

Data Placement and Access. The data bag abstraction alleviates the need for workers to choose the nodes on which to place data, or to redistribute data when a new worker is created. The insert chunk operation on a data bag writes the chunk in a pseudorandom cyclic order across the storage nodes. Spreading the chunks across the storage nodes helps improve insert throughput. Similarly, the remove operation by a worker requests a chunk in a pseudorandom cyclic order across storage nodes. If it does not find a chunk at the node, it tries the next storage node in the cyclic permutation. When a bag contains many chunks that are spread across all the storage nodes, then the first probe will succeed. As the bag gets close to empty, we require more probing.

Storage Load Balancing. Workers can independently request any chunk in a bag from any storage node and they can insert a chunk at any storage node. This decentralized approach reduces latency, but can lead to load imbalance across storage nodes. In the absence of any coordination between workers accessing the storage nodes, these nodes may have significant load imbalance, with some nodes sitting idle, while others are accessed heavily.

Hurricane uses *batch sampling* to ensure high storage utilization. We borrow this idea from decentralized scheduling for data-parallel jobs [31, 38], and recent work on specialized analytics on secondary storage [41]. In this scheme, each compute node sends requests to a fixed number b of different storage nodes concurrently. As these nodes become idle, they respond to a remove request by sending data from the bag to the compute node, or by performing the requested bag insertion.

We choose the number of outstanding requests from a compute node such that for m storage nodes, there are always bm outstanding storage requests. We can easily derive a lower bound on the utilization of each storage node in this setup. The probability that a node is not sent a request is $(1 - \frac{1}{m})^{bm}$. Therefore the expected utilization of a storage node is simply the probability that it is busy, which is the same as the probability that at least one compute node sends a request to it:

$$\rho(b, m) = 1 - (1 - \frac{1}{m})^{bm} \tag{1}$$

The ideal utilization is one (100%). With $b = 1$ outstanding requests, the utilization is at least 63%, with $b = 2$, the utilization is 86%, and with $b = 3$, the utilization is 95%. In practice, we pick $b = 10$, which ensures over 99% utilization even for thousands of storage nodes. Hurricane ensures that at most b concurrent requests are in progress at each compute node. This also serves as a simple flow control scheme to avoid overwhelming the storage nodes.

Batch sampling also helps reduce the latency associated with removing items from bags that are close to empty. This latency is roughly $\frac{m \cdot L}{b}$, where L is the round-trip latency of a single probe operation.

3.4 Adding and Removing Nodes

Hurricane allows dynamic addition and removal of compute and storage nodes from an application. This is easy to support for two reasons: 1) data is stored at storage nodes, separately from compute nodes, and 2) the compute nodes run independently of each other.

The application master can add and remove compute nodes at any point during job execution to accommodate variations in load. A compute node is added by starting the Hurricane framework and configuring the framework with a list of storage nodes, and starting a task manager on the node. A compute node is removed by stopping its task manager after its current workers have completed.

The application master may also add or remove storage nodes during job execution. A storage node is added by starting a Hurricane server on the node. The application master then informs the compute nodes about the new node, allowing workers to place data there. When a storage node is removed, it stops accepting insert requests while still allowing remove requests. When all its bags become empty, the node can be removed.

3.5 Assumptions and Limitations

Since Hurricane spreads data across all storage nodes, without considering locality, an underlying assumption in our design is that the network is not a critical bottleneck in the system. This requirement is met by many analytic workloads and deployments today. Recent work has shown that for many workloads, the network is not the bottleneck, and its effect is mostly irrelevant to overall performance [14]. This is because much less data is sent over the network than is accessed from disk [36]. Our storage system is designed to optimize the latter bottleneck. While our approach increases network communication, network interface speeds today are easily able to keep up with storage bandwidth. A 10 GigE interface can easily support modern disks as well as fast SSDs, and 40 GigE networks are becoming more common. Thus, we expect that network endpoints will not be a bottleneck in our deployments. Similarly, high bisection bandwidth is available at rack scale today, and many clusters are deployed at these scales. For example, in 2011, Cloudera reported a median cluster size of 30 and a mean of 200 nodes [1]. Similarly, many Hadoop clusters have 100-200 nodes [2]. For larger installations, data-center scale full bisection bandwidth networks are being actively researched [33] and deployed [13].

Hurricane does not currently provide a mechanism for speculative execution [18]. We do not attempt to speculatively restart crashed, hung or slow tasks on compute nodes. Crashed or hung tasks will eventually be detected by the application master and will be killed and restarted then. Cloning successfully mitigates stragglers, as slow tasks will eventually be cloned, but this is not done speculatively. We leave the implementation of speculative cloning as future work.

4 IMPLEMENTATION

This section describes the implementation of the various components in Hurricane.

4.1 Task Scheduling

Hurricane minimizes the overhead of task cloning by performing efficient low-latency scheduling through a reliable, distributed task queuing interface called *work bags*. Work bags are similar to data bags and expose the same interface, except they contain tasks, not chunks. Compute nodes remove tasks (including cloned tasks) from work bags to execute on local workers. Similar to data bags, tasks in work bags are distributed across all storage nodes and accessed by compute nodes independently without any single point of control. Unlike traditional scheduling queues, work bags are unordered, allowing for fast decentralized access to their contents.

Each application has three work bags associated with it, a *ready bag*, a *running bag*, and a *done bag*, corresponding to the ready, running, and exited, task states. Compute nodes remove tasks from the ready bag to create workers. Workers execute application code by removing fixed-size chunks from one or more input data bags, computing on the chunks, and then inserting transformed chunks in one or more output data bags. When a worker finishes executing, it inserts its task identifier in the *done* work bag. The application master monitors the done bag and inserts tasks into the ready bag once their dependencies have been completed. The running work bag is used for handling compute node failures.

4.2 Task Cloning

By default, Hurricane runs a single worker for an input bag. At any point, each compute node can signal the application master that it is overloaded and would like a particular task to be cloned to alleviate load. The application master may accept or ignore the cloning request based on a cloning heuristic. We now consider the implementation of overload detection and the heuristic for cloning.

Detecting Overload. A task overload can occur either when the task is CPU bound or IO bound. For detecting a CPU bound task, we need to simply measure the CPU load on the machine. An I/O bound task could be limited by the disk or the network. Since we distribute the data in a bag across storage nodes, a disk-bound task will maximize storage bandwidth, helping us achieve our goal of improving the performance of large datasets with skew. Assuming high bisection bandwidth, a network bottleneck may occur when a node is limited by its endpoint bandwidth. We can detect such a bottleneck by measuring the network throughput at each node. As a result, a compute node generates a clone message periodically, when the CPU or its local network interface is saturated. Currently, we send clone messages at least 2 seconds apart.

Cloning Heuristic. Hurricane uses a simple heuristic to estimate whether cloning a task is worthwhile, using the following quantities: k is the number of clones processing a task, T is the expected time to finish the task without cloning, T_C is the expected time to finish the task with cloning, T_{IO} is the expected additional I/O time due to cloning, i.e. the time to read and load additional task state and the time to merge the clone's output data. It follows that $T_C = \frac{k}{k+1}T + T_{IO}$.

Given the above quantities, Hurricane clones a task if $T_C < T$, i.e. $\frac{k}{k+1} \cdot T + T_{IO} < T$, which simplifies to:

$$T > (k+1) \cdot T_{IO} \tag{2}$$

In other words, cloning is worthwhile when the time to finish the task without cloning is greater than the product of the number of clones and the I/O time resulting from cloning. For example, assume a task is expected to finish in 10 seconds with 4 clones, and the clones are overloaded. Adding a fifth clone brings down the completion time to 8 seconds. So the cloning overhead cannot be more than 2 seconds, or else it will likely delay task completion. The cloning heuristic avoids cloning close to the end of a task, and so we only need a rough estimate of these quantities.

The value of k is known by the application master. T is estimated by sampling the input bag on a few storage nodes to estimate how much data is left and how fast it is emptying. While T_{IO} is application-specific, we estimate it as two times the size of the remaining portion of the input bag that the task will read (for input and output).

4.3 Storage Nodes

Storage nodes provide storage for data bags, and work bags. Data bags bear a resemblance to files stored in distributed file systems like HDFS [43]. In fact, data bags are implemented at each storage node as Linux ext4 regular (buffered) files. A chunk insert request simply appends the chunk to the file associated with the bag. The append operation is atomic, ensuring that concurrent inserts are performed correctly. The insert operations are performed in FIFO order. A remove operation is implemented by reading a chunk from the file sequentially, which increments the file pointer and ensures that the same chunk is never returned again. An end-of-file indicates that all chunks have been removed from this node. The bag API, similar to files, includes other operations, including reusing the contents of a bag, allowing multiple workers to read an entire bag concurrently, sampling the amount of data remaining in a bag, and garbage collecting a bag.

4.4 Fault Tolerance

Fault tolerance is specially important for complex application graphs that must process large amounts of data.

The application master provides a single point of control for the application's execution. This component is application-specific, while all other Hurricane components are application agnostic, and run independently of each other. A consequence of this design is that the crash of a compute node does not interfere or block any other compute node from making progress, since compute nodes are not aware of each other. Similarly, the crash of a storage node does not prevent other storage nodes from serving/storing data. Hurricane applications place all persistent state at the storage nodes, while computes nodes contain only soft state. This approach allows us to use a simple checkpoint-replay mechanism for handling compute node failures, and primary-backup replication for storage node failures.

Application Master Failure. The application master is the only entity that knows about the state of the computation. However, this state is stored in its work bags that are stored on the storage nodes. When the application master fails, we restart it and *replay* the done work bag. Replaying the done work bag involves rereading the entire bag, taking note of each completed task to update the execution graph. Replaying these task completions lets the application master recover the state of the execution graph to its pre-failure state. Once replay completes, the application master resumes normal operation. Neither compute nodes nor storage nodes need to be aware of an application master failure and can continue to execute tasks normally.

Short-lived application master crashes will not usually cause application slowdown as compute and storage nodes can proceed independently of the application master, because the latter is only required to schedule new tasks or to clone existing tasks. Nonetheless, cluster operators may opt to replicate the application master using Apache ZooKeeper [26] for increased resilience to failures.

Compute Node Failure. When a compute node fails, the application master restarts all currently running tasks on the node. To do so, it scans the running work bag for all tasks that the compute node was currently executing. It then terminates all running clones of these tasks. Next, it discards data in the output bags and rewinds the input bags of these tasks, and finally reschedules them by moving them back to the ready bag. From the application's perspective, restarting failed tasks from scratch enables maintaining the exactly once invariant when reading data from input bags.

Our approach is simple to implement and reason about, but comes at the expense of potential slow progress in the presence of many failures. We leave the implementation of a more fine-grained recovery approach to address compute node failures that avoids restarting associated clones as future work.

Storage Node Failure. Hurricane protects the data stored on storage nodes through primary-backup replication. Since data is spread across all storage nodes, an application can tolerate n storage node failures by using $n+1$ replication. Each bag, including data and work bags, is replicated along with bag state, such as the current file pointer position from which the next chunk will be read. The replication level is configurable for each application. In the event of a storage node failure, the application master informs each compute node to use a backup storage node. Compute nodes re-issue requests to the backup storage node and proceed as usual.

Decentralized Control. In the current version of Hurricane, the application master serves as a centralized control plane. On the other hand, the data plane is fully decentralized. While we do not foresee any scalability bottlenecks as a result of this decision, we leave the implementation of a decentralized application control plane for future work.

4.5 Software and Configuration

Hurricane is written in Scala and runs on a standard Java Virtual Machine (JVM) version 8 [3]. The system and all benchmark applications are roughly 7000 lines of code. We use the Akka toolkit (version 2.4) [4] for high performance concurrency and distribution. We use TCP-based netty (version 4) [5] for inter-machine communication, but also support UDP-based Aeron (version 1.2) [6] for high-throughput low-latency messaging. Due to observed instabilities in Aeron, we opted for netty as the default.

We use reasonable defaults for all system parameters. In particular, we do not tune the network stack, the storage subsystem or the JVM.

The chunk size is chosen to minimize the overhead of remote data access, reduce internal fragmentation caused by small bags, and minimize random accesses to disk. Our system uses a 4MB chunk size.

5 EVALUATION

This section evaluates the performance of the Hurricane system and compares it with the Hadoop (version 2.7.4) and Spark (version 2.2.0) systems. Our experiments show the effect of increasing skew and input data size on the system. Then, we evaluate the various design choices we made in Hurricane. Finally, we evaluate the performance of three realistic applications.

We evaluate Hurricane on 32 16-core machines (2 Xeon E5-2630v3), each equipped with 128 GB of DDR3 ECC main memory. The machines have two 6TB magnetic disks arranged in RAID 0. The RAID array sustains a bandwidth of approximately 330MB/s, as reported by fio [7]. The machines are connected through 40 GigE links to a top-of-rack switch which provides full bisection bandwidth. We run no other workload on the machines to ensure that each system can fully utilize the 128GB of main memory. We co-locate compute nodes and storage nodes, and run one storage node per machine using all available storage. We do not enable replication of bags unless explicitly stated.

5.1 Taming Skew

We first evaluate how well Hurricane can deal with skewed workloads along two dimensions: increasing skew in the data, and increasing input data size. To do so, we use the ClickLog application presented in Section 2.1. This application is representative of many analytics workloads, such as the MapReduce paradigm that transforms input data before aggregating it along some dimension.

Input size	320MB	3.2GB	32GB	320GB	3.2TB
Runtime	5.7s	8.9s	22.8s	90s	959s

Table 1: ClickLog runtime over a uniform input (baseline). The total size of the input is scaled from 320MB to 3.2TB of total input.

The input takes the form of text files uniformly distributed across all storage nodes. Each input line contains an IP address. The output is the count of the number of unique IP addresses in each geographic region. We simulate the geolocation function to avoid external API calls.

For the evaluation under skew, we normalize the skew runtimes with the corresponding runtimes for uniform inputs. Table 1 establishes the baseline ClickLog runtimes on uniform inputs with increasing size. We start with 320MB (10MB per machine), and multiply the size by 10 until the input size is 3.2TB (100GB per machine). At 10MB, 100MB, and 1GB per machine, the experiment runs from memory and the performance scales sub-linearly due to execution overhead. The 320GB (10GB per machine) and 3.2TB (100GB per machine) runs execute from disk and scale almost linearly at aggregate disk bandwidth.

To evaluate performance in the presence of skew, we use a synthetic input generator that takes two parameters: *input size* and *skew*. We use a zipf distribution with parameter s ($0 \leq s \leq 1$) to obtain different amounts of skew. Then we generate partitions by dividing the key range into equal parts, so that adjacent keys are placed in each partition.

We show how increasing the skew affects Hurricane. We introduce increasing skew in the input data using skew parameter s, with values 0 (uniform), 0.2 (mild skew), 0.5 (medium skew), 0.8 (medium high skew), and 1 (high skew). The corresponding imbalance between the largest and smallest region is $1\times$, $2.3\times$, $8\times$, $28\times$, and $64\times$.

Given $s = 1$, the largest region makes up 19.6% of the total input. Using Amdahl's law, and assuming that the largest region is the serial (non parallelizable) fraction of the parallel execution and that processing requirements are uniform, we can estimate that the maximum achievable speedup in this scenario is $4.5\times$ when the largest region is not broken up. When using 32 machines, this corresponds to a best case slowdown of $7.1\times$ $(32/4.5)$.

Figure 5 shows Hurricane's slowdown with increasing skew and input sizes. We observe that Hurricane suffers at most $2.4\times$ slowdown across all configurations and significantly less in most cases. By spreading data chunks across all storage nodes and cloning tasks processing large regions to split the work at runtime, Hurricane achieves a much better slowdown than $7.1\times$.

In Figure 5, the normalized runtime increases with increasing skew due to task cloning and merging overheads. Tasks are cloned every two seconds, and so it takes some time until all compute nodes are busy (e.g., in Phase 1). The merge operation introduces overheads

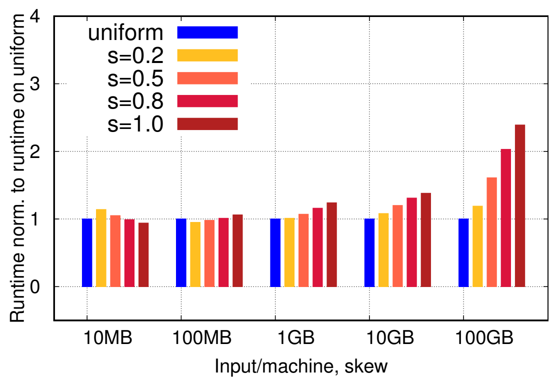


Figure 5: **ClickLog runtime with increasing skew.**

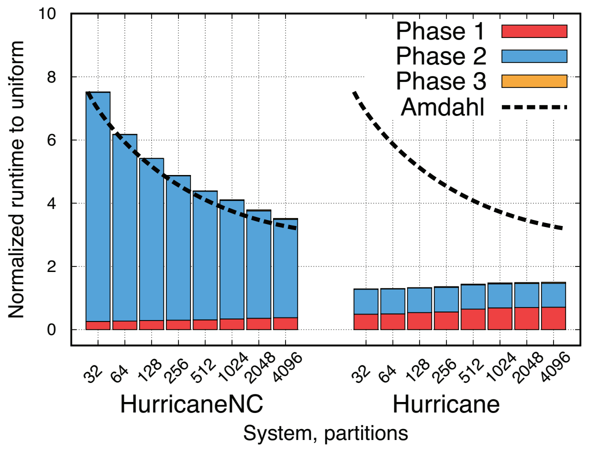


Figure 6: **HurricaneNC (no cloning) and Hurricane with increasing number of partitions on an input of 32GB with skew $s = 1$. Dashed lines represent the best case slowdown using Amdahl's law.**

because it reconciles the partial outputs of clones after their execution. There is no task cloning (and therefore no merging) for the first two input sizes (10MB and 100MB). These experiments run fast due to the small input size, and have little overhead caused by skew. The third input size (1GB) experiences some task cloning in the presence of skew. In the worst case ($s = 1$), this overhead is $0.24\times$, of which 63% is due to cloning delays, and the rest is from merging partial outputs. Experiments for the 10GB input size lead to a significant amount of task cloning for large skew ($s = 0.8$ and $s = 1$). The worst case overhead is $0.38\times$, of which 39% is due to cloning delay in the first phase, and the rest is from merging partial outputs. Figure 9 shows these effects by plotting the sustained throughput over time when the skew s=1, for 10GB input size. Note that as the input sizes becomes larger, the application executes for a longer time, and therefore the relative overhead due to cloning delay decreases.

The largest input size (100GB) suffers from the largest overhead across all experiments, 1.4× for $s = 1$. Unlike smaller input sizes, half of this overhead is due to desynchronized garbage collection pauses at storage nodes, which prevents the system from achieving peak I/O throughput [30]. We are actively looking into this problem, and expect our solution to bring the overhead down to similar levels as that of smaller input sizes.

5.2 Design Evaluation

Varying Partition Sizes. We now evaluate how decreasing the partition size, i.e., creating more tasks of smaller sizes, and scheduling them statically without cloning affects the runtime for a skewed workload. To that end, we run Hurricane with and without cloning (dubbed HurricaneNC) on a 32GB input with skew parameter $s = 1$. We increase the number of partitions from 32 to 4096 so that the average task size decreases with more partitions. The average task size with 32 partitions is 1GB, whereas with 4096 partitions, it is 8MB (comparable to the chunk size).

Figure 6 shows the results for this experiment. We break down the runtimes of each phase. The first phase buckets the IP addresses into regions, the second phase uses a bitset to list unique IP addresses, while the third phase counts the size of the bitset. Hurricane starts with a single worker in Phase 1, which it can clone based on load conditions, while HurricaneNC always runs a single worker per task since it does not clone workers. To ensure a fair comparison for HurricaneNC, we split the Phase 1 input into equal-sized partitions such that each compute node is assigned at least one partition of the input.

There is no skew in Phase 1, and thus the size of tasks has little impact on the phase's runtime. We observe that Hurricane takes a little longer to complete Phase 1 because it starts the phase with a single worker, and cloning on demand introduces some delay for detecting overload. However, the benefit of Hurricane's approach is that the application does not need to specify the correct number of clones. Phase 2 has significant skew and here we observe how cloning reduces the phase's runtime, even though it comes at the expense of a merge. Hurricane can parallelize the processing of large partitions through cloning, whereas HurricaneNC's runtime is dominated by the time it takes to process the largest partition on a single worker. Phase 3 runs very quickly as it does little work.

We plot the best case slowdown as computed in Section 5.1 in dashed lines as a reference. We observe that HurricaneNC's performance closely matches the curve whereas Hurricane clearly stay below. The shape of the results for HurricaneNC indicate that its speedup becomes less significant every time we double the number of partitions until eventually it cannot achieve a better runtime.

We conclude from these results that smaller partitions alone are insufficient for addressing skew: even though the average partition size decreases, large partitions remain comparatively large. In the absence of cloning a single worker must process the largest partition sequentially, and so the system cannot fully leverage the presence of more tasks to achieve better load balancing. Finally, we observe that creating too many small partitions introduces scheduling and storage overheads, as evidenced by the increase in runtime for Phase 1 for both systems.

Cloning and Spreading. We now seek to evaluate which feature of Hurricane works best to address skew, and in particular whether both cloning and spreading data across storage nodes are necessary for good performance. We only present the runtime for the first two phases, since the third phase runs for a short time.

We consider four configurations of Hurricane with different features turned off:

- Configuration 1: *Cloning Off, local data*. Cloning is disabled. We create one task per bag, i.e. one task for Phase 1 and r tasks for Phase 2 (where r is the number of regions). Phase 1 task input is on local disk and its output data is written locally. Phase 2 tasks read their input data from remote machines in parallel.
- Configuration 2: *Cloning Off, spread data*. Cloning is disabled. We create the tasks as before. All data (including initial input) is spread.
- Configuration 3: *Cloning On, local data*. Cloning is enabled. We create one task per bag as before, but the system can clone both Phase 1 and Phase 2. Data is placed as in Configuration 1.
- Configuration 4: *Cloning On, spread data*. Cloning is enabled, as in Configuration 3. All data (including initial input) is spread.

We run the ClickLog application on 8 machines in each of the above four configurations with 80GB of input data (10GB per machine). Figures 7 and 8 show the results for Phase 1 and Phase 2 respectively. Phase 1 is not impacted by skew since each IP is geolocated and placed in the corresponding region bag independently. We observe that spreading data in the bag is essential for good performance as local data places the burden of serving that data on a single storage node. For instance, with local data, turning cloning on only speeds up Phase 1 by 25% because, even though the output of clones is placed on local storage, one machine must still supply the entire input. Figure 8 shows that Phase 2 is severely impacted by skew, as shown in the first configuration. Spreading the data improves performance by 33% (second configuration) because it helps achieve better storage load balance, allowing the machine processing the heaviest region to use all disks when it is the last task remaining, effectively increasing its storage bandwidth. Cloning with local data (configuration 3) is slower than cloning with data that is spread (configuration 4) because clones are introduced with a delay, hence the output is not uniformly distributed across all nodes if it is kept local. Finally, we observe that cloning has the most impact with increasing skew since it allows the heaviest region to be simultaneously processed by multiple workers.

Locality. One might wonder whether Hurricane takes a performance hit by spreading data uniformly in the absence of skew. As we can see from Figures 7 and 8, this is not the case in our deployment because the network is fast enough to match storage bandwidth. As a result, remote bandwidth is roughly the same as local bandwidth.

Overload Detection & Cloning Heuristic. Hurricane clones tasks to rebalance load and increase parallelism for large tasks, allowing the system to better utilize both CPU and storage resources. We evaluate the effectiveness of our overload detection mechanism and cloning heuristic with ClickLog running on 32 machines with 320GB input. We set the skew parameter to 1 (high skew).

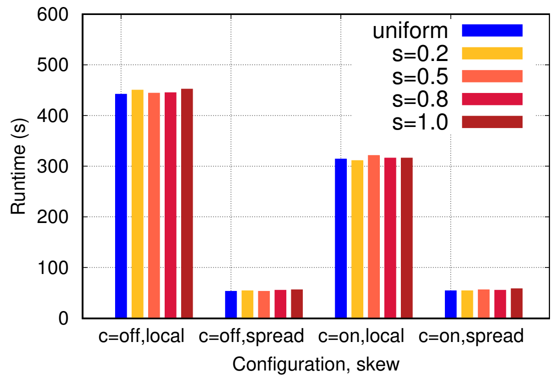


Figure 7: Runtime of ClickLog Phase 1 for different configurations with various features turned off.

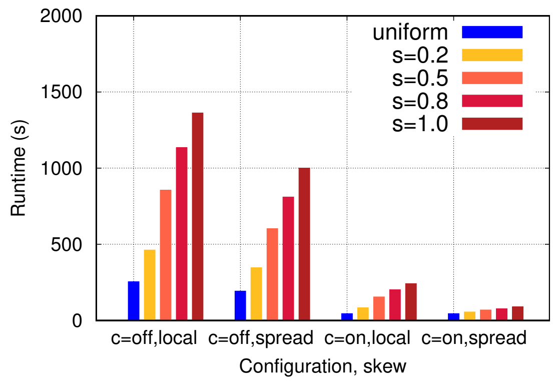


Figure 8: Runtime of ClickLog Phase 2 for different configurations with various features turned off.

Figure 9 shows the aggregate throughput achieved by all compute nodes in the system sampled at one-second intervals. Phase 1 starts with one worker executing the single task. Since the task is CPU bound, it clones rapidly until all 32 machines are running clones around the 15 second time point (the number of clones doubles approximately every 2 seconds). There is no merge in Phase 1, and all workers complete roughly at the same time.

Phase 2 then starts with one task per region, which together occupy all available worker slots at compute nodes. As tasks associated with small regions complete, their associated Phase 3 tasks are scheduled and executed. When they finish, some compute nodes become idle because there are no more available tasks, allowing compute nodes processing larger regions to get higher storage bandwidth. This overloads their CPU, so they issue cloning requests to the application master, which grants them on a case-by-case basis. Eventually, only the largest region remains, with 26 workers simultaneously processing it. Cloning stops beyond 26 workers because storage, and not the CPU, becomes the bottleneck. As this region gets close to the end, the application master rejects further cloning requests, as the merge overhead would become larger than the benefits of cloning. Once

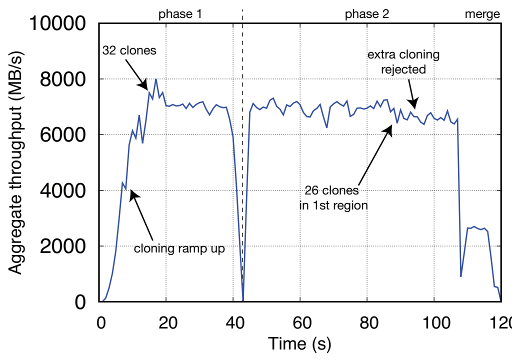


Figure 9: ClickLog throughput over time on 32 machines. The vertical dashed line separates Phase 1 from Phase 2.

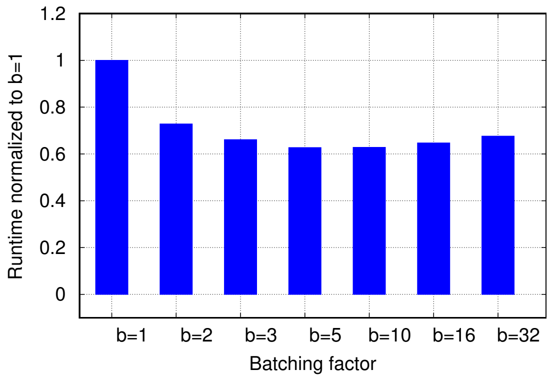


Figure 10: Runtime of ClickLog Phase 1 on 32 machines for various batching factors.

the region is processed, the outputs of each clone are combined by a merge task, and application execution terminates.

It is worth pointing out that throughput remains nearly constant for Phase 2 in spite of significant skew because the system clones tasks on idle nodes when storage is not the bottleneck.

Batch Sampling. We consider the batch sampling technique presented in Section 3.3 and evaluate its impact on performance. Batch sampling of chunks within bags performed by workers is a means for ensuring that storage nodes remain busy throughout their execution and that workers are not starved for data, essentially overlapping computation and communication through prefetching of chunks.

We consider Phase 1 of ClickLog with various batching factor values, from $b = 1$ (i.e. one chunk at a time) to $b = 32$ (i.e. one in-flight request per storage node). Figure 10 shows that allowing workers to prefetch multiple chunks is essential for good performance and to keep storage nodes busy. However, prefetching too many chunks ($b = 32$) is undesirable since it risks overwhelming storage nodes and could lead to unfairness. $b = 10$ is the sweet spot, allowing us to achieve 33% runtime improvements simply through better overlapping of computation with storage I/O.

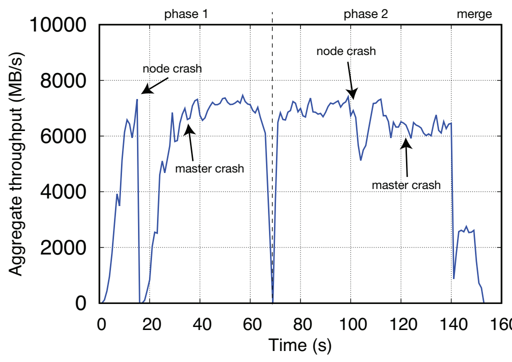


Figure 11: ClickLog throughput over time on 32 machines with worker and application master crashes. The vertical dashed line separates Phase 1 from Phase 2.

Throughput and Storage Utilization. Hurricane storage nodes are designed to scale storage I/O throughput observed by compute nodes with increasing number of storage nodes. We verify that this is the case by running a synthetic benchmark where each worker writes a fixed amount of random data (100GB) and then reads the data back. We start on one machine and then double the number of machines until 32, thus doubling the aggregate amount of data stored in storage nodes. The results indicate that Hurricane sustains maximum I/O bandwidth, regardless of the number of machines involved. For instance, we achieve 330MB/s read bandwidth with one machine and 10.53GB/s read bandwidth with 32, an increase of 31.9× for 32× more machines. Similarly, we achieve 327MB/s write bandwidth with one machine and 10.39GB/s write bandwidth with 32, i.e. 31.7× speedup. By increasing the number of storage nodes, applications can scale throughput while maintaining high storage throughput.

Fault Tolerance. We evaluate the impact of compute node and application master crashes on throughput. Figure 11 shows the aggregate throughput over time for an execution of ClickLog on a 320GB input using 32 machines (10GB per machine). We forcibly crash a compute node twice: once during phase 1 and once during phase 2. In each case, we also crash the application master 20 seconds after recovering from the compute node crash. Compute node crashes cause throughput to deteriorate temporarily, as the system must stop all corresponding task clones and restart the crashed task. Since phase 1 consists of a single task, the crash of the compute node requires restarting all workers in the system. In phase 2, the same crash only requires restarting all associated clones of the task (recall, different regions have different tasks), and thus the throughput only degrades by $\sim$ 25%. Application master crashes have little impact on throughput for two reasons: the master's recovery is extremely fast (less than 1 second), and once tasks are placed in the work bag, compute nodes can proceed independently of the master's status.

5.3 Applications and Comparisons

Finally, we consider three workloads, representative of real-world applications, described below. We compare the performance of Hurricane on these workloads with optimized implementations in Hadoop and Spark.

- *ClickLog*: count the distinct number of occurrences of each IP address per region in a log of clicks on advertisements. This application was presented in Section 2.
- *HashJoin*: given two relations and an equality operator between values, for each distinct value of the join attribute, return the set of tuples in each relation that have that value. This is a classic problem in relational databases.
- *PageRank*: execute 5 iterations of the PageRank algorithm [39] on a large real-world power-law graph. PageRank has many real-world applications and is a well-known benchmark used in graph processing systems. This is a multi-stage application.

ClickLog. We compare our ClickLog results with Hadoop and Spark by evaluating each system's performance under different levels of skew.

The implementation of ClickLog in Hadoop maps parts of the input text to the workers, which tokenize it, parse the IP addresses, geolocates the IP address per region, and output intermediate lists of IP addresses in each region. The reduce phase goes over the intermediate lists to perform a distinct count. Spark operates in much the same way on resilient distributed datasets, mapping the input to workers, tokenizing, geolocating by region, and counting distinct IP addresses. Wherever possible, we use the same data structures and perform the same operations for all implementations. In particular, all implementations use bitsets to perform the distinct count.

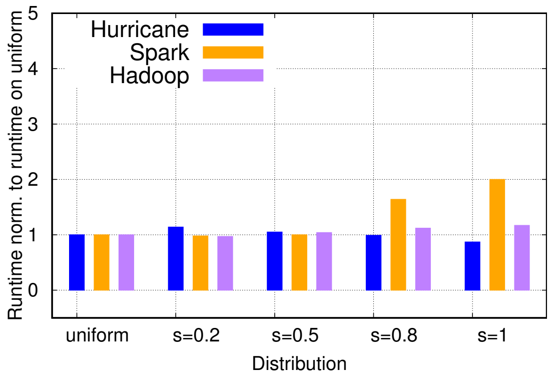
We use HDFS [43] in the case of Hadoop and Spark. We make sure that both Hadoop and Spark read their input data from the local disk and write the much smaller output without replication. We also split the job into enough tasks to ensure that Hadoop and Spark can utilize all available cores in the cluster and have enough opportunities to balance load. We try multiple values for the number of partitions (ranging from 100 to 10000) and report the best runtime across all configurations. We verify that no task was restarted because of a crash during the execution.

Table 2 shows the runtime of all three systems on uniform inputs for two input dataset sizes. The 320MB input is guaranteed to fit in memory on a single machine even in the presence of high skew, while the 32GB may not fit in a single machine due to Java runtime overheads. All three systems (in particular Hadoop) experience some overhead when executing on the small 320MB input as a result of small task sizes.

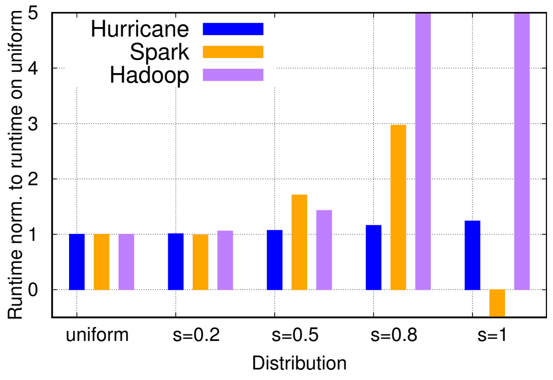
Hurricane achieves lower overall runtimes because it does not need to sort intermediate data, which allows better overlap between computation and communication. Both Hadoop and Spark must sort intermediate data to ensure key ranges do not overlap. However, eliminating sorting in Hurricane does not come free, since the system must perform an additional merge task for cloned tasks.

System	320MB	32GB
Spark	8.2s	32.4s
Hadoop	37.1s	50.3s
Hurricane	5.7s	22.8s

Table 2: ClickLog runtime over a uniform input for 320MB and 32GB input sizes.



(a) 320MB



(b) 32GB

Figure 12: Comparison of Hurricane, Spark and Hadoop when the skew is increased for input sizes 320MB and 32GB. A full bar indicates that the execution did not terminate in under an hour and was forcibly terminated. Negative bars indicate a crash.

Figure 12 shows the slowdown on all three systems as skew is introduced in the input. To ensure a *fair* comparison, the runtime for each system are normalized to its *own* runtime with the uniform input.

We observe that both Hadoop and Spark suffer significant performance degradation in the presence of skew, particularly as the input size increases. Spark runs out of memory and crashes with highly skewed tasks due to a hard limitation of 16GB placed on task memory. Hadoop suffers from a large increase in runtime due to the impact of skew on a few reducers, forcing them to spill.

System	3.2GB ⋈ 32GB		32GB ⋈ 320GB	
	s=0	s=1	s=0	s=1
Hurricane	56s	89s	519s	1216s
Spark	81s	1615s	920s	>12h

Table 3: HashJoin runtime for two different relation sizes and different amounts of skew. $s = 0$ **is uniform.**

System	RMAT-24	RMAT-27	RMAT-30
Hurricane	38s	225s	688s
GraphX (Spark)	189s	3007s	> 12h

Table 4: Comparison of Hurricane and GraphX on 5 iterations of Pagerank over an RMAT-27, RMAT-30, and RMAT-32 graph.

HashJoin. Table 3 shows the runtimes for two joins, one between a small 3.2GB relation and a larger 32GB relation, and the second between a 32GB relation and a 320GB relation, for both Hurricane and Spark. For both joins, we introduce skew in the first (smaller) relation, causing a much larger hit rate for some keys. The join in Hurricane splits the smaller relation into 32 equal-sized partitions, and sorts them in memory. It then creates 32 corresponding partitions in the larger relation, and finally streams the larger partitions, while the smaller partition is in memory, outputting matching keys. Spark's implementation proceeds in a similar fashion but with more partitions to make sure all available CPU cores are used. We try varying number of partitions (ranging from 100 to 10000) and report the best overall runtime. As before, we ensure that input data is read from the local disk and that there are no task crashes during execution. We also disable output replication.

We can observe that Spark struggles with skew due to load imbalance and that this effect worsens as the input size increases. The slowdown is directly caused by a larger hit rate in some partitions. Hurricane handles the situation more gracefully due to its ability to spread both input and output across storage nodes as well as its ability to clone the tasks containing keys with larger hit rate.

PageRank. Table 4 compares the runtime of PageRank in Hurricane and Spark's GraphX, a state-of-the-art graph-parallel library for graph applications. We compare on different input sizes using 32 machines. We use the RMAT graph generator [15] to generate real-world power-law input graphs, i.e. graphs whose degree distribution is skewed. RMAT-24 has 16 million vertices and 256 million edges, RMAT-27 has 128 million vertices and 2 billion edges, while RMAT-30 has 1 billion vertices 16 billion edges.

PageRank is computed iteratively for 5 phases after an initialization phase. In each phase, each vertex in the graph sends its current PageRank along outgoing edges to neighboring vertices, and then aggregates the PageRanks received by neighbors along incoming edges to compute its new PageRank. PageRank is essentially a *scatter* of vertex values performed by joining vertex identifiers with outgoing edge source vertex identifiers, followed by a *groupby* aggregation on vertex identifiers. Because this is an iterative algorithm with changing input data, it is representative of long multi-phase application graphs. We use GraphX's example PageRank implementation for comparison, ensure the input is read locally, and check that no crashes occur during execution.

As we can see, Hurricane performs much better than GraphX on all input sizes. We observe significant task cloning in Hurricane throughout the execution, particularly for partitions with high-degree vertices, which allows each stage of the computation to finish in a timely fashion. GraphX struggles to finish executing on larger input sizes due to spilling and shuffling overhead. These results demonstrate that Hurricane handles skew effectively in multi-stage applications.

6 RELATED WORK

Adaptive Partitioning of Work. As far as we can tell, Hurricane is the first cluster computing framework to adaptively partition work based on load observed by workers during task execution. This design is made possible through fine-grained data sharing among multiple workers executing the same task and programmer-defined merge procedures.

Several techniques have been proposed to split analytics jobs into smaller tasks in order to mitigate skew and improve load balance. These techniques require manual intervention from the programmer and are application- and input-specific. For instance, they require fine-tuning the programmer-defined split function [35], exploiting commutativity and associativity to combine identical keys[46], and/or splitting records for the same key across multiple partitions[8]. Hurricane mitigates skew in an application-independent manner by dynamically splitting partitions when a task is cloned. We have shown that our approach mitigates skew effectively, without requiring tuning of the application for specific data sets, and that it is applicable to arbitrary operations, such as finding unique values.

Traditional cluster computing frameworks split data into partitions and use shuffling and sorting to merge them back in an application-independent way [25, 32, 47]. This often times comes at the cost of sorting intermediate outputs, and prevents records with the same key being sent to multiple reducers, which can cause load imbalance in the presence of skew. More importantly, this approach places constraints on the shape of partitions, making it harder to redistribute a partition in a balanced way. Hurricane takes a different approach by empowering application developers to provide a custom merge method, when applicable. This merge subsumes the traditional shuffling and sorting paradigm, while being more flexible, because it allows the outputs of clones that have been created at any point during execution to be merged in an application-specific manner.

Although adding a merge procedure to existing frameworks is relatively simple, taking full advantage of it would require significant re-engineering and changes to the execution model to allow for tasks to be repartitioned on-the-fly. Fault tolerance mechanisms would also need to be adapted to account for the possible presence of multiple partial outputs. Finally, frameworks which rely on key sorting to send records to the appropriate reduce may also end up losing the ability to combine records by key as a result of such changes.

Skew Mitigation. SkewTune [28] mitigates skew in MapReduce programs by identifying slow tasks and repartitioning them to run on idle nodes. Since the system is intended to be a drop-in replacement for MapReduce, it suffers similar limitations, namely that the output order must be preserved and the data placed locally on the original worker. While this approach can help with skew, it also causes significant data movement, which can overwhelm already overloaded workers. SkewTune can also worsen performance inadvertently by repartitioning tasks that are close to completion.

Camdoop [17] performs in-network data aggregation during the shuffle phase of MapReduce applications, which can help mitigate data skew by decreasing the amount of data moved and the overall load on the network. Unfortunately, this solution requires special hardware that is not currently available. We believe such hardware would also benefit Hurricane deployments.

Straggler tasks are a challenge for analytics workloads [12]. A commonly used method for handling stragglers is speculative execution, which involves detecting a straggler as soon as possible and restarting a copy of the task on another machine [11]. While this approach helps with machine skew, it does not address data or compute skew. Hurricane allows slower workers to split their task via cloning, avoiding the need to restart the task from scratch.

Garbage collection (GC) can be a major cause of skew for applications written in garbage-collected languages such as Java, Scala, or Python. GC induces uncoordinated pauses across JVM [37], thereby reducing overall throughput and increasing tail-latency. Recent research attempts to mitigate this problem by synchronizing (or desynchronizing) garbage collection across all workers running the same application to minimize unpredictability [30]. Hurricane is also prone to GC pauses, but its decentralized design and finer-grained partitioning help reduce its impact.

When analytics applications suffer from skew in their input or intermediate data, they may be forced to spill data to disk because it does not fit in memory, leading to serious performance issues. Spongefiles [20] allows machines with large datasets to use the memory of remote machines as a backing store, thereby avoiding spilling. Hurricane spreads data by default across all machines through the bag abstraction. Since bags are backed by files, the spreading of data helps even when the dataset size does not fit in the main memory of the entire cluster, because it allows spreading the disk I/O.

Joins. Load balancing for parallel joins has been extensively studied in parallel parallel databases. Earlier work [21, 34] focused on careful partitioning based on input sampling to achieve load balance, while more recent approaches [42] use late binding to gain flexibility and reassign partitions to other workers. Hurricane requires less focus on partitioning, relying instead on cloning and merging for handling skew.

Distributed Scheduling. Support for scheduling tiny tasks has led to the design of distributed or hybrid schedulers such as Sparrow [38] or Hawk [19]. Sparrow uses a batch sampling algorithm to schedule tasks, whereas Hawk partitions the cluster for large and small jobs, and uses a randomized work stealing algorithm to place short jobs. Hurricane also recognizes the need for efficient scheduling, in particular for clones, and schedules tasks in a distributed and decentralized way through work bags.

Storage Disaggregation. Hurricane draws inspiration from recent research in storage disaggregation [14, 27, 33]. Decoupling storage and computation makes it possible to achieve better utilization and balance across workloads, the cost being remote storage access. This cost is minimal for small-to-medium clusters with sufficient bisection bandwidth. Hurricane takes these ideas a step further by always spreading data across machines in the cluster, even when the data fits in main memory.

7 CONCLUSIONS

This paper makes the case for Hurricane, a system for high-throughput analytics designed from the ground up for handling skewed workloads. Hurricane works well because it is designed to dynamically partition work based on load imbalance. It allows programmers to seamlessly write applications whose performance does not degrade significantly in the presence of skew. Applications using Hurricane benefit from high capacity and scalability, as well as inherent load balance and high parallelism.

We anticipate that with the coming data deluge there will be a need for analytics over large volumes of data with a non trivial amount of skew, for example to test hypotheses and discover patterns. In this context, we believe it is critical for application developers to have tools that enable them to focus on writing high-value business application code rather than spending time fine-tuning partitioning and system parameters to obtain good performance.

Acknowledgments: We would like to thank our anonymous reviewers, shepherd Dushyanth Narayanan, Calin Iorgulescu, Amitabha Roy, and Christoph Koch for their feedback that improved this work. We would also like to thank Florin Dinu for his invaluable feedback, his last minute printer debugging skills, and for motivating us to keep improving our implementation. This work was supported in part by the Swiss National Science Foundation NRP 75 Grant No. 167157.

REFERENCES

[1] 2011. http://www.dbms2.com/2011/07/06/petabyte-hadoop-clusters/. (2011).
[2] 2018. https://wiki.apache.org/hadoop/PoweredBy. (2018).
[3] 2018. http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html. (2018).
[4] 2018. http://akka.io/. (2018).
[5] 2018. https://netty.io/. (2018).
[6] 2018. https://github.com/real-logic/Aeron. (2018).
[7] 2018. http://freecode.com/projects/fio. (2018).
[8] 2018. http://cgnal.com/blog/using-spark-with-hbase-and-salted-row-keys/. (2018).
[9] 2018. Stack overflow. (2018).
[10] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. 2012. Tarazu: optimizing mapreduce on heterogeneous clusters. In *ACM SIGARCH Computer Architecture News*, Vol. 40. ACM, 61–74.
[11] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones.. In *NSDI*, Vol. 13. 185–198.
[12] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the Outliers in Map-Reduce Clusters using Mantri.. In *OSDI*, Vol. 10. 24.

[13] Alexey Andreyev. 2014. Introducing data center fabric, the next-generation Facebook data center network. *74145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network* (2014).

[14] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The end of slow networks: It's time for a redesign. *Proceedings of the VLDB Endowment* 9, 7 (2016), 528–539.

[15] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining*. SIAM.

[16] Graham Cormode and S. Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-min Sketch and Its Applications. *J. Algorithms* 55, 1 (2005), 58–75.

[17] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. 2012. Camdoop: Exploiting in-network aggregation for big data applications. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 3–3.

[18] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[19] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. 2015. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference*. USENIX Association, 499–510.

[20] Khaled Elmeleegy, Christopher Olston, and Benjamin Reed. 2014. Spongefiles: Mitigating data skew in mapreduce using distributed memory. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 551–562.

[21] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and adaptive online joins. *Proceedings of the VLDB Endowment* 7, 6 (2014), 441–452.

[22] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms 2007 (AofA07)*. 127–146.

[23] Christos Gkantsidis, Dimitrios Vytiniotis, Orion Hodson, Dushyanth Narayanan, Florin Dinu, and Antony IT Rowstron. 2013. Rhea: Automatic Filtering for Unstructured Cloud Storage.. In *NSDI*, Vol. 13. 2–5.

[24] Benjamin Gufler, Nikolaus Augsten, Angelika Reiser, and Alfons Kemper. 2012. Load balancing in mapreduce based on scalable cardinality estimates. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 522–533.

[25] Apache Hadoop. 2009. Hadoop. (2009).

[26] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8. 9.

[27] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 29.

[28] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2012. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 25–36.

[29] Jimmy Lin and others. 2009. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce. In *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, Vol. 1.

[30] Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. 2015. Trash Day: Coordinating Garbage Collection in Distributed Systems.. In *HotOS*.

[31] Michael Mitzenmacher. 2001. The Power of Two Choices in Randomized Load Balancing. *Trans. Parallel Distrib. Syst.* 12, 10 (2001).

[32] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 439–455.

[33] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. 2012. Flat Datacenter Storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 1–15.

[34] Alper Okcan and Mirek Riedewald. 2011. Processing theta-joins using MapReduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 949–960.

[35] Kay Ousterhout, Aurojit Panda, Josh Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. 2013. The Case for Tiny Tasks in Compute Clusters.. In *HotOS*, Vol. 13. 14–14.

[36] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun, and V ICSI. 2015. Making Sense of Performance in Data Analytics Frameworks.. In *NSDI*, Vol. 15. 293–307.

[37] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun, and V ICSI. 2015. Making Sense of Performance in Data Analytics Frameworks.. In *NSDI*, Vol. 15. 293–307.

[38] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 69–84.

[39] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[40] Smriti R Ramakrishnan, Garret Swart, and Aleksey Urmanov. 2012. Balancing reducer skew in MapReduce workloads using progressive sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 16.

[41] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 410–424.

[42] Lukas Rupprecht, William Culhane, and Peter Pietzuch. 2017. SquirrelJoin: network-aware distributed join processing with lazy partitioning. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1250–1261.

[43] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. IEEE, 1–10.

[44] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, and others. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 5.

[45] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2.

[46] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.

[47] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. *HotCloud* 10, 10-10 (2010), 95.

[48] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments.. In *Osdi*, Vol. 8. 7.