Pangea: Monolithic Distributed Storage for Data Analytics

Jia Zou
Rice University
jiazou@rice.edu

Arun Iyengar
IBM T. J. Watson Research Center
aruni@us.ibm.com

Chris Jermaine
Rice University
cmj4@rice.edu

ABSTRACT

Storage and memory systems for modern data analytics are heavily layered, managing shared persistent data, cached data, and non-shared execution data in separate systems such as a distributed file system like HDFS, an in-memory file system like Alluxio, and a computation framework like Spark. Such layering introduces significant performance and management costs. In this paper we propose a single system called Pangea that can manage all data—both intermediate and long-lived data, and their buffer/caching, data placement optimization, and failure recovery—all in one monolithic distributed storage system, without any layering. We present a detailed performance evaluation of Pangea and show that its performance compares favorably with several widely used layered systems such as Spark.

PVLDB Reference Format:
Jia Zou, Arun Iyengar, Chris Jermaine. Pangea: Monolithic Distributed Storage for Data Analytics. *PVLDB*, 12(6): 681-694, 2019.
DOI: https://doi.org/10.14778/3311880.3311885

1. INTRODUCTION

One of the defining characteristics of modern analytics systems such as Spark [57], Hadoop [54], TensorFlow [7], Flink [9], and others [6, 22] is that they tend to be heavily layered. For example, consider Spark. It caches job input/output and execution data (for shuffling and aggregation) in two separate memory pools: a storage pool (that is, the RDD cache or DataFrame) and an execution pool. These software components, in turn, are implemented on top of the Java virtual machine (JVM), which constitutes its own software layer. Since application output is transient and does not persist, such data must be written to *yet another* layer, an external storage system such as HDFS [14], or additionally cached in an external in-memory system like Alluxio [39] or Ignite [2].

The Perils of Layering. Layering allows for simpler components and facilitates mixing and matching of different systems that use compatible interfaces. However, it also introduces significant costs in terms of performance. Some important factors leading to high costs are:

(1) Interfacing Overhead. Data needs to be repeatedly de-objectified (serialized), copied, and objectified (de-serialized) as it is moved between layers, which wastes CPU cycles and memory resources. For this reason, we have found that an external cache (such as Alluxio or Ignite) while important for persisting and sharing data across applications, can incur a more than 50% overhead compared to Spark without such a cache, when running the widely used machine learning algorithm k-means.

(2) Data Placement Complexity. When job input data must be stored at all layers, redundant copies of the same data object will be maintained at each layer. Our observation is that in the k-means workload, about 30% of total required memory for processing is used for storing redundant input data.

In addition, layering can result in the same data object being copied multiple times *within* the same layer. For example, at the storage layer, HDFS replicates each data block for high availability. At the application layer, each Spark application might load and partition all of the objects in a dataset in a different way using different partition sizes and partition schemes. In a layered system, there is obviously a knowledge gap between storage and applications. It may make more sense to bridge the knowledge gap by removing the layers and allow the storage system to offer multiple partitionings that can be used as replications for failure recovery and also re-used at the application layer. We observe up to 20× speedup for TPC-H queries using and re-using multiple partitionings in the storage layer.

(3) Un-Coordinated Resource Utilization. In addition, moving seldom-accessed, in-RAM data to secondary storage (paging it out) is a fundamental tool for dealing with large datasets. However, when no single system component has a unified view of the system, it is difficult to make good paging decisions: one component may page out data when it needs more RAM, while there is a far less important piece of data in another component that should have been paged out. For example, although HDFS can utilize Linux OS's `fadvise` and `madvise` system calls [44] to specify file access patterns for optimizing paging in OS buffer cache, it has no way to influence allocation of memory resources within tools like Alluxio, Ignite, and Spark. We find that when size of the working set exceeds available memory, good paging decisions can achieve $1.8\times$ to $5\times$ speedup compared with LRU, MRU, and DBMIN [21] policies for the k-means workload.

Pangea: A Monolithic Distributed Storage Manager. The benefits of layering—including more flexible, and interchangeable tools and components—will outweigh the costs for many applications. For example, if a distributed file system does not have the right recovery model for a given application, one can (in theory) swap in a new file system that addresses the problem. However, for applications that demand high performance, layering's performance cost compared to a monolithic architecture may be unacceptable.

Monolithic architectures are not new, and the perils of heavily modular systems are well-understood; indeed, performance vs. flex-

Proceedings of the VLDB Endowment, Vol. 12, No. 6
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3311880.3311885

ibility concerns underlie the classical microkernel vs. macrokernel operating system debate [15,40,47]. But in practice, for Big Data analytics systems, the layered, modular approach has won out. Our goal in this paper is to re-examine this debate in the context of distributed analytics. We seek to synthesize 30 years of ideas in storage management and distributed analytics into a single system—which we call Pangea[1]—and to examine how well that system competes with layered alternatives. As we will show, Pangea compares favorably with layered systems in terms of performance.

Pangea represents an important alternative in the distributed analytics system design space. Developers of high-performance distributed systems need not feel compelled to accept the costs associated with layering, whether the goal is developing a high-performance distributed machine learning platform such as TensorFlow [7], a distributed version of a machine learning pipeline such as scikit-learn [48], a Big Data computing platform such as our own PlinyCompute [60], or high-performance distributed query or SQL engine such as Impala [36]. If performance is key, a monolithic base such as Pangea should be considered as an alternative to building on a layered storage system, and also as an alternative to the all-too-commonly-chosen option of developing a distributed storage system from scratch to achieve the best performance.

Some of the key ideas embodied by Pangea are:

1. *Different types of data should be managed simultaneously by the storage system and different types of data should be handled differently.* Thus, we redefine the locality set abstraction in classical relational systems [21] to enable the storage manager to simultaneously manage different data durability requirements, lifetimes, and access patterns at runtime. We describe a paging strategy that utilizes that information to optimize page replacement decisions based on a dynamic priority model that utilizes the locality set abstraction.

2. *Data placement and replication should be integrated within the storage system.* In distributed storage systems such as HDFS, data are replicated for fault tolerance; data are then typically replicated *again* at higher levels with different data placement schemes for application-specific processing. In Pangea, data may be replicated using different partitioning strategies, and then such information is available to an external query optimizer so that a particular copy will be chosen at runtime. Pushing this functionality into a single monolithic system means that applications can share and re-use such physical organizations, and it obviates the need to store even more copies to facilitate recovery for node failures.

3. *Computational services should be pushed into the storage system.* Thus, Pangea ships with a set of *services* that provide an application designer with efficient implementations of batch processing operations such as sequential read/write, shuffle, broadcast, hash map construction, hash aggregation, and so on. Moving these services inside of Pangea allows job input/output data and ephemeral execution data buffered/cached all be managed within the same system. Those services can also bridge the knowledge gap between storage and applications so that storage can understand various application-specific performance implications for paging and data placement.

Our Contributions. Key contributions are:

- We describe the design and implementation of Pangea, a monolithic distributed storage system, designed to avoid the problems of heavily layered systems. Pangea is implemented as more than $20,000$ lines of C++ code.
- Pangea presents a novel storage design that consolidates data that would typically be stored using multiple, redundant copies within multiple layers into a locality set abstraction (redefined based on DBMIN [21]) within a single layer. A locality set can be aware of rich data semantics and use such information for multiple purposes, such as replication and page eviction.
- We conduct a detailed performance evaluation and comparison of Pangea to other related systems. For k-means, Pangea achieves more than a six times speedup compared with layering-based systems like Spark. For the TPC-H benchmark, Pangea achieves up to a twenty times speedup compared with Spark. For various Pangea services, Pangea achieves up to a fifty times speedup compared with related systems.

2. RELATED WORK

Pangea is a monolithic system that encompasses many different functionalities: a distributed file system, a memory management system, and various distributed services. Many recent papers and projects have examined these topics in the context of Big Data and data analytics.

Distributed file systems (DFSs), such as the Google File System [30], the Hadoop Distributed File System [14], Microsoft Cosmos [19], and IBM GPFS-SNC [31] provide scalable and fault-tolerant persistent file storage. Distributed Object Storage (DOS) systems such as Amazon S3 [1], Google Cloud Storage [3], Microsoft Azure Cloud Storage [16], OpenStack Swift [12], Ceph [53] also provide storage for persistent objects. These typically provide simple operations such as `select` and `aggregate` which resemble Pangea's services, but as Pangea is meant to be a general-purpose substrate for building distributed analytics systems, its operations (shuffle, hash aggregation, join map construction, etc.) tend to be more substantial.

Many existing systems include similar functionality to that offered by Pangea, though as a monolithic framework, Pangea includes a wider range of functionalities in a single system and subsumes these narrower systems. For example, in-memory file systems such as Alluxio [38,39] can be deployed on top of DFS and DOS to allow disk file blocks or objects to be cached in memory and accessible to different upper-layer cluster computing systems. Ignite [2] can store Spark data as SharedRDDs and cache it. The built-in memory management in frameworks such as Spark (with RDD cache [57], Datasets/DataFrames [5,11], and extensions such as Deca [41]) is responsible for loading and caching input data. Workload-aware storage systems, such as BAD-FS [13], also include a subset of Pangea's functionalities. BAD-FS includes a job scheduler that makes workload-specific decisions for consistency, caching, and replication. Data for each workload is managed separately and most of optimizations are for the cluster-to-cluster scenario in a wide-area network.

Among its other components, Pangea includes a paging system. There has been extensive work on page replacement algorithms such as LRU-K [46], DBMIN [21], LRFU [37], MQ [59] and so on. As we will describe in Sec. 3.2 in detail, Pangea borrows and extends the idea of *locality sets* from DBMIN. Other algorithms and systems mainly consider recency, reference distance, frequency, and lifetime, which is effective when processing a single type of data as in a traditional relational database buffer pool or RDD cache, but can be less efficient for managing multiple types of data in Big Data analytics, such as a large volume of intermediate data produced during computations (for example, during hash aggregations and joins). GreedyDual [18,56] is a widely used cache replacement framework which associates a numerical value reflecting the desirability of keeping an object in cache. Objects are kept in cache or replaced based on these numerical values. The independent reference model

[1] Pangea is used as the storage system of PlinyCompute [60].

(IRM) [26] and its extensions [28, 29, 32, 55] can model cache references to different pages in order to estimate the hit/miss ratio, which is orthogonal with this work and can be applied to model and evaluate our proposed approach in the future.

There has been work on understanding access patterns of applications and mapping them to the right storage layer. `fadvise` and `madvise` allow users to specify file access patterns. Self-learning storage systems such as ABLE [24, 43] predict access patterns based on generic file system traces.

Pangea also manages multiple partitionings and uses those for distributed replication and failure recovery. Some efforts consider data partitioning in object-oriented systems (CoHadoop [25] and Hadoop++ [23]) and others in SQL-based systems (SCOPE [58], SQLServer [8], DB2 [49], and so on). These mainly rely on standalone failure recovery mechanisms: replicating each block to several nodes [14], using erasure codes to store parity blocks [51], and so on. So each partitioning may incur additional redundancy in terms of replicated blocks or parity blocks. C-Store [52] is a column store for relational data that maintains K-safety; multiple projections and join indexes for the same data are maintained, so that K sites can fail and the system can still maintain transactional consistency.

3. OVERVIEW OF PANGEA

Pangea is designed to manage all data—both intermediate and long-lived data, and their buffer/caching, and placement—in a monolithic storage system. We begin by detailing the fundamental problem that makes devising a unified, “Swiss army knife” system difficult: the disparate data types with different properties that an analytics system will encounter.

3.1 Disparate Data Types and Key Properties

In the context of buffer pool management and file caching, devising fair and efficient policies for allocating memory among multiple competing datasets is always a difficult problem [17, 21]. Unfortunately, the problem is *even more difficult* in the context of analytics processing, due to the fact that there are more types of data that the system may need to manage (consider the simple example of k-means clustering, as illustrated in Fig. 1).

In analytics processing, data can be categorized into following: (1) *User data*, which are the ultimate input and output of batch processing applications. (2) *Job data*, which are short-lived intermediate data resulting from transformation pipelines running over user data. Then there is intermediate data that exists for a short time within the transformations, as they are executed, which includes (3) *Shuffle data* that are moved across workers (such as between map workers and reduce workers in a MapReduce job) and (4) *Hash data* that consist of a hash table and key-value pairs, as used in hash-based aggregation and join operations. Other types of short-lived intermediate data exist as well—those are generally termed *execution data*. Key differences in those data types (and also in different datasets of the same type) are in following properties:

Durability requirements. User data are read repeatedly by various applications, so they need to be persisted immediately once created (we call this a `write-through` requirement). However, job data and execution data are transient, intermediate data that do not require persistence. If transient data are being evicted from memory when their lifetime hasn’t expired, they need to be spilled to disk (which we call `write-back`). `write-back` data should generally have a higher priority for being kept in memory than `write-through` data, because evicting write-back data incurs additional I/O cost.

Data lifetimes. In iterative computations, the input data are needed across multiple iterations. Shuffle data may span two job stages (e.g. map and reduce), while aggregation data live as long as an aggregation is being performed. Data that will not be accessed should be evicted as soon as their lifetimes expire.

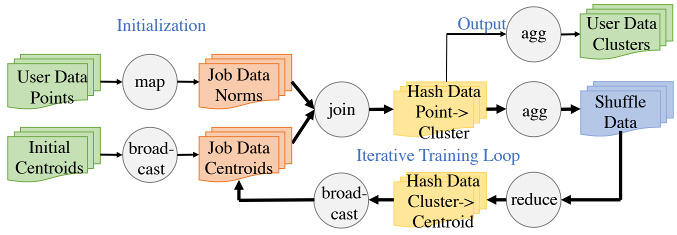


Figure 1: A k-means Example. Even a computation as simple as k-means produces user data, intermediate job data, as well as execution data (the local vector of centroids that must be shuffled, for example).

Current operations. Evicting data that have just been written should be avoided if other things are equal, because such data tend to be read soon. For example, in a query execution flow that consists of multiple job stages, the input and output of a job stage may have similar durability requirements, lifetime, access recency, and so on, but the output is more likely to be accessed by next job stages than the input.

Access patterns. Different operators and their associated data types often exhibit disparate access patterns. For example, a map operator processes each data element in the input dataset, so eviction of data elements that haven’t yet been processed should be avoided. In contrast, an aggregation operator needs to insert each element of input data into a hash table, which is often randomly accessed, and should have high priority to stay in memory.

Temporal recency. Datasets that have been accessed more recently are more likely to be reused, e.g. recurring services [20, 33, 34].

3.2 The Locality Set Abstraction

In a layered system, data within a layer (e.g. a user file system, or a memory pool in computing platform) tend to be relatively homogeneous. For example, HDFS and Alluxio only store user data for various applications; the Spark RDD cache stores user data and job data only for the current application that is in processing; the Spark execution memory pool only stores execution data for the current application. Pangea’s task is more difficult, because it manages all data within a single layer—data exhibiting different access patterns, data sources, and lifetimes are present.

To facilitate management of disparate data types, we borrow and update the *locality set* abstraction, which was first proposed in DBMIN [21] as the query locality set model (QLSM) for buffer management in classical relational systems. QLSM mainly defines a set of access patterns; and for each access pattern, there is a predefined eviction policy such as MRU, LRU, and so on, and also a predefined algorithm to derive the desired size of locality set, where a locality set is defined as the set of buffer pool pages for a file instance. DBMIN’s buffer pool management strategy is based on the desired size and eviction policy of each locality set. When the size of a locality set is larger than its desired size, one or more pages will be chosen and evicted based on the eviction policy associated with this locality set.

While DBMIN’s locality set idea serves as a building block for Pangea, DBMIN was first proposed more than 30 years ago for transaction processing, at a time when modern analytics workloads did not exist. As such, some of DBMIN’s basic assumptions need to be updated. Most problematic is the implicit assumption in DBMIN

Table 1: Some Locality Set Attributes.

Attribute	Supported values
`DurabilityType`	`write-back`, `write-through`
`WritingPattern`	`sequential-write`, `concurrent-write`, `random-mutable-write`
`Location`	`pinned`, `unpinned`
`ReadingPattern`	`sequential-read`, `random-read`
`Lifetime`	`lifetime-ended`, `alive`
`CurrentOperation`	`read`, `write`, `read-and-write`, `none`
`AccessRecency`	`sequence id for last access`

that all locality sets correspond to persistent database tables. As such, DBMIN does not consider durability requirements and assumes all data should be persisted to disk—a fine assumption in transaction processing, but not reasonable for modern analytics workloads that frequently produce large volumes of transient intermediate data. DBMIN requires a maximum number of buffer pages (i.e. desired size) on a per file-and-query basis to be known (or guessed) beforehand, which is again not reasonable for intermediate datasets produced as the result of running opaque user-defined functions on data, as is standard in modern data analytics. If the total number of estimated buffers exceeds available memory, new requests are blocked, which is again unreasonable on modern analytics workloads.

Thus, we update some of the ideas in DBMIN. A Pangea locality set is simply a set of pages (or blocks) associated with one dataset that are used by an application in a uniform way and are distributed across a cluster of nodes.

In Pangea, there is no hard partitioning of the buffer pool to the different locality sets (which, for transient data created via UDFs, would require solving difficult estimation problems, such as guessing the size of the locality set). All Pangea locality sets share the same buffer pool (see Sec. 5) and unified eviction policy (see Sec. 6).

In Pangea, all pages in one locality set must have the same size, which can be configured when creating the locality set. Data organization in a page is flexible, and each page can represent a chunk of relational data, or a container (e.g. vector, hash-map and so on) of objects. Pages from a locality set may be stored on disk. However, unlike DBMIN, it is not necessary for each page of a locality set to have an image in an associated file. In Pangea, a locality set page can reside in disk, in memory, or both, because transient data (job data and execution data) are also stored in locality sets; such locality sets may have had only a fraction (or none) of their pages on disk.

Depending on the application requirements, different locality sets must be managed differently. To achieve this, unlike DBMIN, Pangea locality sets are given a set of *tags*, or attributes, either by an application or automatically by inference. The various attribute categories and their supported values are listed in Table 1. The attributes generally correspond to the key factors as identified in Sec. 3.1.

In analytics processing, there are typically three writing patterns: `sequential-write` where immutable (write-once) data is written to a page sequentially; `concurrent-write` where multiple concurrent data streams are written to one page (write-once), e.g. to support creation of shuffle data; `random-mutable-write` where data can be dynamically allocated, modified, and deallocated in a page (write-many), e.g. to support aggregation, pipeline processing, or join. There are two reading patterns: `sequential-read` for data such as shuffle data; and `random-read` for data such as hash data.

Determining attributes. Pangea provides various *services* to read and write locality sets. In Pangea, attributes such as `ReadingPattern`, `WritingPattern` and `CurrentOperation` are automatically determined at runtime through invocations of services, since each service exhibits specific writing and reading patterns. For example, the sequential read and write services exhibit the `sequential-read` and the `sequential-write` pattern, respectively. The shuffle service exhibits the `concurrent-write` pattern, and the hash service exhibits both the `random-mutable-write` and `random-read` patterns. Thus, when (for example) an application associates the locality set with a sequential allocator that provides the sequential write service, then `WritingPattern` must be `sequential-write` and `CurrentOperation` must be `write`; if an application associates the locality set with a sequential iterator that provides the sequential read service, then `ReadingPattern` must be `sequential-read`, and `CurrentOperation` must be `read`.

Here are some examples of using Pangea locality sets.

```
//create a set
LocalitySet myData = createSet("data");
//add single object (sequential write)
myData.addObject(myObject);
//add a vector of objects (sequential write)
myData.addData(myVec);
//sequential read
vector<PageIteratorPtr> * iters =
  myData.getPageIterators(numThreads);
for (int i = 0; i < iters->size(); i++) {
  // to start worker threads to read pages
  runWork(iters->at(i), myWorkFunc);
}
//create a set for storing shuffled data
LocalitySet shuffledData = createSet("shuffled");
//invoke shuffle service in one worker
while((PagePtr page = iter->next())) {
 ObjectIteratorPtr objIter
   = getObjectIterator(page);
 while((RecordPtr record = objIter->next())) {
   PartitionID partitionId =
     hash(udfGetKey(record));
   VirtualShuffleBufferPtr buffer = shuffledData.
     getVirtualShuffleBuffer(workerId, partitionId);
    buffer->addObject(record);
  }
}
```

More code examples for using locality set and invocation of services are presented in Sec. 7 and Sec. 8.

Heterogeneous replication. In a monolithic system like Pangea, a locality set can have multiple replicas to do double-duty and facilitate *both* recovery and computational savings due to the ability to provide multiple physical data organizations. Furthermore, replicas in Pangea are visible and usable by all applications running on top of Pangea. Applications or a data placement optimizer can apply different partition schemes and page sizes to a locality set to generate new locality sets, and register those replicas with the same replica group. This replica and partitioning information (along with other useful attributes of the data) is stored in Pangea's statistics database and made available to all applications with appropriate permissions that are running on top of Pangea for choosing a replica for computation.

Details about data recovery using heterogeneous replicas are described in Sec. 7.

3.3 System Architecture

The Pangea system has five components, which we describe at a high level now and will be described in more detail subsequently.

(1) The distributed file system. In Pangea's file system, a file instance is associated with one locality set and consists of a sequence of pages that are on-disk images of all, a portion of, or even no pages from the associated locality set. Those pages can be distributed across disks in multiple nodes.

(2) The buffer pool. Unlike other distributed storage systems, on each node Pangea utilizes a unified buffer pool to manage both user data and execution data. The buffer pool manages most or all of the RAM that is available to Pangea and the applications that are running on top of Pangea, in the form of a large region of shared memory. The idea is that applications rely on Pangea to collectively manage RAM for them.

(3) The paging system. The paging system is responsible for evicting pages from the buffer pool to make room for allocating new pages. It maintains a dynamic priority model that orders locality sets according to their durability requirements, lifetimes, access patterns, access recency, and so on. For each locality set, a paging strategy is automatically selected based on its access patterns. When Pangea needs more RAM, it finds the locality set with the lowest priority and uses its selected strategy to evict one or more victim pages from the locality set. Details are described in Sec. 6.

(4) Distributed data placement system. Each locality set can have multiple replicas which allow for both recovery and computational efficiency. Each replica may have different physical properties, so that the replica with the best physical organization can be selected for a given computation. The data placement system manages the partition, replication, and recovery of locality sets and is described in Sec. 7.

(5) Distributed services. To realize the benefits of Pangea's unified storage architecture, applications need to entrust all of their datasets (including user data, job data, shuffle data, aggregation data, and so on) to Pangea. To facilitate this, Pangea provides *services* to the applications that run on top of Pangea. The *sequential read/write service* allows each of multiple threads to use a sequential allocator to write to a separate page in a locality set. Each worker can then use a provided concurrent page iterator to scan a subset of pages in the locality set. The *shuffle service* allows multiple writers to write to the same page using a concurrent allocator, so that multiple data streams for the same shuffle partition can be combined into a single locality set. The *hash service* allows a locality set to be allocated as a key-value hash table, through a dynamic allocator. Details are discussed in Sec. 8.

Deployment and Security Considerations. The Pangea distributed system consists of one light-weight manager node responsible for accepting user applications, maintaining statistics and etc., and many worker nodes that run the functionalities of above five components.

For cloud deployment, Pangea ensures security by delegating authority to remote processes through the use of public-key cryptography. We require that the users must be assigned a valid public-private key pair when deploying the cluster. Then the user must submit the private key when bootstrapping the system. Under the hood, in the initialization stage, the Pangea manager node relies on this user-submitted private key to access workers for collecting system information. A non-valid key will cause the whole system to terminate.

4. THE DISTRIBUTED FILE SYSTEM

In keeping with our "no layering" mantra, to avoid redundant data copying between a cache layer (e.g. Spark RDD, Alluxio) and a file system layer (e.g. the HDFS or the OS file system), on each worker node, a Pangea process contains a user-level file system which uses the Pangea buffer pool (described in Sec. 5) to buffer reads and writes. All reads/writes are implemented via direct I/O to bypass the OS buffer cache.

In Pangea, a distributed file instance that is associated with one locality set is implemented using one Pangea data file and one Pangea metadata file on each worker node. On each worker node, a Pangea data file instance is chunked into fixed size pages. Depending on user-selected settings, a Pangea data file instance can be automatically distributed across multiple disk drives on its worker node. The set of pages allocated to each disk drive can be mapped to a physical disk file. The Pangea metadata file is simply a physical disk file used to index each page's location and offset.

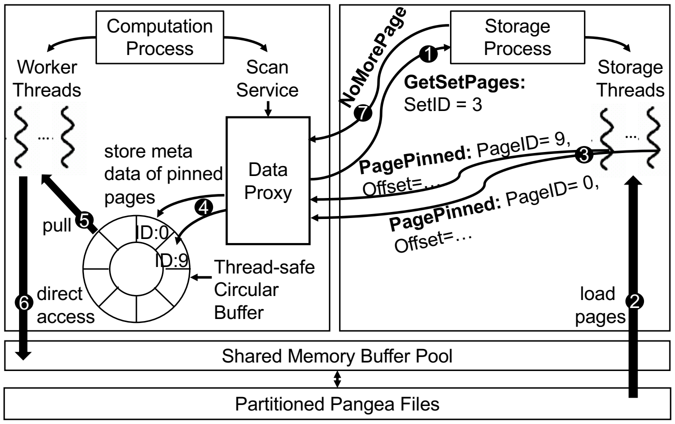


Figure 2: Data scan: long living worker threads pull page meta data from a circular buffer and access page data through shared memory.

A centralized Pangea manager shared by the distributed file system and other distributed components manages locality set metadata (such as database name, dataset name, page size, data attributes, partition scheme, replica group and so on). Metadata for each page are stored in the Pangea metadata files at each worker node. Compared to HDFS (where locations are stored for each block at the name node) the Pangea manager stores considerably less metadata.

When reading a page, Pangea first checks the buffer pool to see whether the page is already cached in memory. If the page is not present in the buffer pool, the page needs to be cached first. When writing, depending upon a locality set's `DurabilityType`, there are two durability settings available: (1) `write-through`, where each page, once written, will be cached and also persisted to disk; and (2) `write-back`, so that a dirty page is first cached in the buffer pool, and written to disk only if there is no free space in the buffer pool and the page has been chosen to be evicted. User data are often configured as write-through, while transient job data and execution data are often configured as write-back.

5. THE BUFFER POOL

Separate buffering mechanisms partitioned across the various software layers found in today's analytics installations make it difficult for the various buffering softwares to coordinate and assign resources in an optimal way. To sidestep this, Pangea caches all data in one unified buffer pool that is used by all applications utilizing Pangea.

On each node, Pangea uses the anonymous `mmap` system call to allocate a large chunk of shared memory as a buffer pool in a storage process. The computation process that has multiple worker threads accessing the data concurrently is forked by Pangea and given access to the shared memory.

Those computation threads coordinate read/write access to the buffer pool with Pangea via a data proxy based on sockets, but the actual data served are communicated via shared memory. In this way, data written to a buffer pool page are visible to computations immediately, without any copy and moving overhead. A significant fraction of the typical serialization/deserialization overhead present in current analytics systems can also be avoided for accessing local, in-memory data.

All information required to access and manage a buffered page, such as the the page offset in shared memory, is communicated through the network. We implement various message-based communication protocols directly on top of TCP/IP for the computation framework to exchange page location information with Pangea's storage facilities.

Taking sequential read, as shown in Fig. 2, as an example, the data proxy in the computation process first sends a `GetSetPages` message to the storage process. Then the storage process starts multiple threads `pin`ning the pages in the locality set to be scanned, and the metadata (e.g. relative offset in the shared memory) for each page that is `pin`ned to the buffer pool is sent to the computation process via socket. The data proxy then puts those metadata into a circular buffer that is thread-safe, from which multiple computation threads can fetch the meta data for a page at a time, and access that page through shared memory for running computations.

Therefore, in the computation process, the threading model supported by Pangea, is quite different from the "waves of tasks" model used in Spark and Hadoop where a task thread will be scheduled for each block of data, and many such small tasks will execute concurrently as a "wave".

Instead, in applications and computation frameworks built on Pangea, when executing a job stage, the computation process starts multiple worker threads, which are long living and do not terminate until all input data for the job stage have been executed. Then in a loop, each worker thread pulls page metadata from the concurrent circular buffer as described above.

Therefore, in Pangea, there is no need to consider the "all-or-nothing" property of "wave"-based concurrent task execution, trying to have all or none of the inputs for tasks in the same wave cached, as proposed in memory caching system for data-intensive parallel jobs such as PACMan [10].

To write output to a page, the data proxy sends a `PinPage` message to the storage process, which then pins a page in the buffer pool, and sends back the metadata of the page to the computation.

The buffer pool on each node uses a dynamic pool-based memory allocator to allocate pages with various sizes from the same shared memory pool. Pangea supports two main pool-based memory allocators: the Memcached slab allocator [27] and the two-level segregated fit (TLSF) memory allocator [42]. We use TLSF by default because it is more space-efficient for allocating variable-sized pages from the shared memory.

A hash table is used to map the page identifier to the page's location in the buffer pool. Each page cached in the buffer pool can be mapped to a page image in one locality set. Once the page must be spilled, the page will be appended to the locality set's associated file instance.

As mentioned, each page has a `pinned`/`unpinned` flag. In addition, each page has a `dirty`/`clean` flag to indicate whether the page has been modified. Page reference counting is used to support concurrent access. A service can `pin` an existing page in a locality set for reading and writing. Once the page is cached in the buffer pool, its flag is set to `pinned`, with its reference count incremented.

Once the processing on a page has finished, the service `unpins` the page and the reference count is decremented. When the buffer pool has no free pages to allocate for an incoming pinning request, the paging system will evict one or more unpinned pages and recycle their memory. Before evicting an unpinned page that is marked as "dirty" but is still within its locality set's lifetime, we need to make sure that all changes are written back to the Pangea's file system first. We describe paging in more detail in the next section.

6. DATA-AWARE PAGING SYSTEM

Given a page pinning request when there are no free pages in the buffer pool, the paging system will select a locality set whose next page-to-be-evicted has the lowest priority and ask that locality set to evict one or more pages based on this locality set's selected paging strategy. Each locality set has its own paging policy, chosen to match its attributes. Pangea selects MRU as the paging strategy for locality sets labeled `sequential-write`, `concurrent-write` and `sequential-read`, and LRU as the paging strategy for locality sets labeled `random-mutable-write` and `random-read`.

The number of pages evicted from the selected locality set is based on the `CurrentOperation` locality set attribute. If the selected locality set is labeled `write` or `read-and-write`, then a single victim page is selected, as evicting data that has just been written should be avoided, as explained in Sec 3.1. For read-only locality sets, 10% of the locality set is evicted, as we have found that in Big Data analytics, read-only operations tend to be well-behaved, with a lot of temporal locality. Once an application has not read one page in a while, it is unlikely to use any pages from the set for awhile, and a larger eviction is warranted.

The key question is how the priority of each locality set's next page-to-be-evicted's priority is computed. Assuming for a moment that each locality set has only pages that are alive (`lifetime-ended` is false), we choose a locality set to be the victim locality set if the next page-to-be-evicted from the locality set has the lowest expected cost compared to the priority of the next page-to-be-evicted from all other locality sets, within a given time horizon t.

There are two parts to this expected cost: the cost c_w is the cost to write out the page, and the cost c_r to read it again, if necessary. Then, the overall expected cost of evicting a page is:

$$c_w + p_{reuse} \times c_r$$

where p_{reuse} is the probability of accessing the page within the next t time ticks. That is, no matter what, we pay a cost for evicting the page, and we may pay a cost associated with reading the page if it is re-read at a later time.

c_r can be estimated as $c_r = v_r \times w_r$ where v_r is the profiled time to read the page from disk; w_r represents the penalty associated with the reading pattern of a locality set. Reading spilled data for the sequential read pattern only requires reading the page to memory, so $w_r = 1$. But reading spilled data that has a random reading pattern (`random-read`) incurs a higher cost ($w_r > 1$), because reading such data requires reconstruction of the hash map and re-aggregation of the spilled data.

c_w is determined by the locality set's durability requirement and can be computed as $c_w = \frac{d}{v_w}$. v_w the time to write the page to disk (also collected via profiling) and $d = 1$ for the `write-back` requirement, and $d = 0$ for `write-through`.

p_{reuse}, which is the probability that the page is reused in the next t time ticks, is a bit more complicated to estimate. p_{reuse} for a page is computed from the page's λ value, where λ is the rate (per time tick) at which the page is referenced. If we model the arrival time of the next reference to each page as a Poisson point process [35], then the probability that the page is referenced in the next t time ticks is $1 - e^{-\lambda t}$ (this follows from the cumulative density of the exponential distribution, which models the time-until-next arrival for a Poisson point process).

There are a number of ways that λ for a page can be estimated. We can collect the number of references n to the page in the last t' time ticks, and estimate the rate of references per time tick as $\lambda \approx n/t'$. This quantity is a bit difficult to deal with in practice, however. It requires storing multiple references to each page, maintained over a sliding time window.

λ can also be estimated from the time since the last reference, which is what we use in our Pangea implementation. If a page was last referenced at time tick t_{ref} and the current time tick is t_{now}, the number of references since the beginning of time can be estimated as $t_{now}/(t_{now} - t_{ref})$; dividing by t_{now} to get the number of references per time tick gives $\lambda \approx 1/(t_{now} - t_{ref})$; that is, λ is the inverse of the time-to-last reference of the page.[2]

Finally, note that the previous discussion assumes that `lifetime-ended` is false for each locality set. If there are one or more locality sets where `lifetime-ended` is true, these are always chosen for eviction first, again according to the minimum expected cost of evicting a page from the locality set.

A note on rate vs. probability. There is a strong relationship between using p_{reuse} computed via an exponential distribution with a time horizon of $t = 1$, and simply weighting a page's read cost c_r by λ (the inverse of the time since last reference in the case of Pangea). In fact, the latter is a linear approximation to the former. If one approximates the exponential computation of p_{reuse} with a linear function (a first-degree Taylor series approximation of the exponential function about the point $\lambda' = 0$), we have:

$$\begin{aligned} p_{reuse} &= 1 - e^{-\lambda t} \approx 1 - e^{-\lambda' t} + te^{-\lambda' t}(\lambda - \lambda') \\ &= 1 - e^{0} + te^{0}\lambda = t\lambda = \lambda \end{aligned}$$

7. DATA PLACEMENT

Pangea's monolithic architecture allows it to use replication to perform double-duty: to provide for fault tolerance (as in HDFS), *and* to provide for computational efficiency by allowing for multiple physical data organizations.

Appropriate data partitioning (such as co-partitioning of related data on the same join key) can avoid shuffling across the network, and speed up operations such as joins by many times [25].

While systems such as Spark provide similar functionality, a partitioned RDD in Spark is specific to computation and will be discarded once an application runs to completion; it can not be reused for future runs of applications. Although a Spark developer could materialize the repartitioned dataset as a different HDFS or Alluxio file, there are two shortcomings: (1) HDFS can not recognize and utilize those files for failure recovery; (2) the process of loading data into RDD cache is controlled by Spark task scheduler, which is optimized for locality but doesn't guarantee locality, thus a repartition stage at runtime is still needed before performing a local join that can be pipelined with other computations. As a result, a Spark application often invokes `repartition()` to tune the split size and then applies `partitionBy()` to tune the partitioning scheme every time the application is executed.

In Pangea, such physical data organizations are persistent and can be shared across applications. For example, a *source set* `lineitem` can be partitioned into a *target set* `lineitem_pt`:

```
registerClass("LineItem.so");
LocalitySet myLineItems = getSet("lineitem");
LocalitySet myReplica =
  createSet<LineItem>("lineitem_pt");
PartitionComp<String, Lineitem> partitionComp =
  PartitionComp(getKeyUdf);
partitionSet (myLineitems, myReplica, partitionComp);
```

This code creates a partition computation that extracts a `String`-valued key from each `Lineitem` and uses that key to partition the set of `myLineItems`. Then, a user registers `lineitem_pt` as a replica of lineitem using a `registerReplica()` API:

```
registerReplica( myLineItems, myReplica,
 numPartitions, numNodes, partitionComp);
```

Now, any application that finds the partitioning useful can use this new replica to perform computations.

Under the hood, the source set and the target set are placed in the same *replication group*. By definition, each set in a replication group contains exactly the same set of objects organized using a different physical organization, and an application running on top of Pangea can choose any appropriate set in the replication group, based on the desired physical properties of the set. Registering multiple sets in a replication group in this way has the added benefit of obviating the need to store multiple copies of each object on different machines in order to allow for failure recovery.

However, having the sets in a replication group do "double duty" in this way requires some care. Because the various physical organizations are chosen for computational reasons (pre-partitioning based upon a join key to make subsequent joins faster, for example) all copies of an object may just happen to be stored on the same machine. We call such objects "colliding" objects. Colliding objects are a problem because if the machine holding the colliding objects fails, the objects are lost.

In practice, however, the number of colliding objects is small. If a transformation to a target set is random (hashing, for example), then the expected number of colliding objects can be estimated as n/k, for n objects and k worker nodes. In our experiments using real partitioning schemes, the number of colliding objects is small. When partitioning the TPC-H lineitem table that has 5.98 billion `Lineitem` objects (about 79GB in size) using two partitioning schemes onto ten Pangea worker nodes (on `l_orderkey` and `l_partkey` respectively), there are 53.39 million colliding objects in total. When we partition the same `Lineitem` table to 20 nodes, there are only 15 million colliding objects. When we further use 30 nodes for the same partitioning, we find no colliding objects.

Given that the number of colliding objects is relatively small, to achieve a complete recovery of lost data, we identify and record all colliding objects at partitioning time. Then those colliding objects will be stored in a separate locality set, and replicated using an approach similar to HDFS replication.

The recovery process first requires calculating the key range for all lost partitions from the failed node. Then, to recover a particular replica (referred to as the *target replica*), the system arbitrarily selects another replica from the replication group (the *source replica*). The system runs the target replica's partitioner on the source replica to extract the key for each object in the source replica. If a key falls in the range of lost partitions, the key and associated object are buffered and later dispatched to the location where the associated key range in the target replica is being recovered. At the same time that the source replica is being processed, all colliding objects from the replication group whose instance in the target replica have been placed on the failed node are recovered. These objects are recovered by processing the special locality set used to store the colliding objects from the replication group.

The strategy can be extended to handle concurrent r-node failures by separately replicating any object of which the replicas are located on fewer than $r + 1$ nodes. This requires significantly more disk space (i.e. if partitioning is random, the expected ratio of such objects in k-node cluster is $(1 - \frac{k \times (k-1) \times ... \times (k-r)}{k^{r+1}})$). Fortunately, since modern analytics frameworks are usually deployed

[2] The inverse of the page's reference distance can also be seen as yet another reasonable estimate for λ, as this effectively replaces $t_{now} - t_{ref}$ with the page's last observed between-reference time as an estimate for the expected time interval between page references. We choose the time since last reference, however, as it requires only a single reference to be valid.

on smaller clusters ranging in size from a few to a few dozen nodes, concurrent multiple-node failures are the exception [6, 22].

8. SERVICES

Pangea provides a set of services to enable various types of locality sets to be cached in one buffer on each worker node and their attributes to be learned at runtime. We now describe a few of the services offered by Pangea.

The Sequential Read/Write Service. This service allows one or more threads on each worker node to read or write data to or from a locality set. To write to a locality set sequentially, a worker first needs to configure the locality set to use a sequential allocator to allocate bytes from the page's host memory sequentially for writing byte-oriented data. If a page is fully written, the storage will unpin the page and pin a new page in the locality set.

To scan a locality set using one or multiple threads on a worker node, the application first needs to obtain a set of concurrent page iterators from the locality set and dispatch each iterator to a thread, using code like the following:

```
LocalitySet myInput = getSet(setId);
//if "write-back" is not specified here,
// "write-through" is used by default.
LocalitySet myOutput =
  createSet(setName, "write-back");
//if "sequential" is not specified here, the dynamic
//secondary allocator will be used by default.
myOutput.setAllocationPolicy("sequential");
vector<PageIteratorPtr> * iters =
  myInput.getPageIterators(numThreads);
for (int i = 0; i < iters->size(); i++) {
  // to start worker threads
  runWork(iters->at(i), myOutput, userfunc);
}
```

Then in each thread, we sequentially write to pages pinned in `myOutput`:

```
while((PagePtr page = iter->getNext())) {
  ObjectIteratorPtr objectIter
      = createObjectIterator(page);
  while((RecordPtr record = objIter->next())) {
    myOutput.addObject(userfunc(record));
  }
}
```

Virtual Shuffle Buffer/Shuffle Service. For shuffling, all data elements dispatched to the same partition need to be grouped in the same locality set, and we create one locality set for each partition.

It is important to allow multiple shuffle writing threads to write data elements belonging to the same partition to one page concurrently, to reduce batch latency and memory footprint. Thus, we use a secondary, small page allocator that first pins a page in the partition's locality set, then dynamically splits small pages (of several megabytes) from a large page, and allocates small pages to multiple threads. Once all small pages are fully written, the small page allocator unpins this page and allocates a new page from the buffer pool for splitting and allocating small pages.

To allow threads to access small pages transparently, we offer a virtual shuffle buffer abstraction. Each shuffle writer allocates one virtual shuffle buffer for each partition. A virtual shuffle buffer contains a pointer to the small page allocator that is responsible for the partition's locality set, and also the offset in the small page that is currently in use by its thread. Then, each partition's locality set can be read via the sequential read service.

An example of the user shuffle code has been described in Sec. 3.2.

Virtual Hash Buffer/Hash Service. Pangea's hash service adopts a dynamic partitioning approach, where each page contains an independent hash table, as well as all of its associated key-value pairs. We implement this by using C++ STL unordered-map along with the Memcached slab allocator [45] to replace the STL default allocator. The Memcached slab allocator uses the current page as its memory pool, so all memory allocation is bounded to the memory space hosting that page. Each page is a hash partition, and all hash partitions are grouped into one locality set.

We start from K pages as K root partitions, all indexed by a virtual hash buffer. When there is no free memory in one page, we allocate a new page from the buffer pool and split a new child hash partition from the partition in the page that has used up its memory. We iterate using this process until there is no page that can be allocated from the buffer pool to construct a new hash partition. Then, when a page is full, the system needs to select a page, `unpin` it, and spill it to disk as partial-aggregation results.

When all objects are inserted through the virtual hash buffer, we re-aggregate those spilled partial aggregation results for each partition. User code is as follows:

```
//by below API, "write-back" and "dynamic"
//allocation policy will be automatically inferred
VirtualHashBufferPtr<string, int> buffer
  = createVirtualHashBuffer(myOutput);
while((RecordPtr record = myInput.next())) {
  string key = udfGetKey(record);
  int value = udfGetValue(record);
  if( buffer->find(key) == nullptr ) {
    buffer->insert(key, value);
  } else {
    buffer->set(key, value);
  }
}
```

Pangea also provides other services such as *join map service* for building a distributed hash table from shuffled data; and *broadcast map service*, which broadcasts a locality set and constructs a hash table from it on each node for implementing broadcast join. Due to space limitation, we omit the details here.

9. EVALUATION

In this section, we evaluate Pangea. We test applications such as k-means clustering and the TPC-H benchmark in a distributed cluster, and perform a detailed performance analysis of various Pangea services in a single node.

For the distributed benchmark, we use 11 to 31 AWS r4.2x large instances, where each instance has eight cores, 61GB memory, and a 200GB SSD disk. For running micro-benchmarks of various services, we use one AWS m3.xlarge instance that has four CPU cores, 15GB memory and two SSD instance store disks. On all machines we install Ubuntu 16.04 and use Spark 2.3.0, Hadoop 2.7.6, Ignite 2.6.0, and Alluxio 1.7.1.

9.1 Distributed Benchmark

We have argued in the introduction that a monolithic system such as Pangea should be considered as an option for building high-performance data analysis tools. Flexibility may suffer, but the performance may be excellent.

To investigate whether this claim is reasonable, we implement two computations directly on top of Pangea and compare the performance with a more conventional layered approach: implementing the computation on top of Spark, which is itself using HDFS or another storage system, all of which is running on top of the JVM.

The first benchmark is a simple k-means computation, which is a widely used benchmark for evaluating the effect of storage, because one of the main challenges in k-means is data locality: keeping as much data in memory as possible [57].

For the second benchmark, we actually implement a distributed relational query processor on top of Pangea. This is particularly interesting because we are implementing a reasonably complicated tool on top of Pangea, which makes it possible to quantify the effort of implementing such a tool as illustrated in Tab. 2, as well as the performance benefit.

Table 2: Source Code Break-down for a Pangea-based Relational Query Processor.

Component	SLOC
Scan	35
Join	1545
Build broadcast hash map	161
Build partitioned hash map	270
Aggregate: local stage	117
Aggregate: final stage	465
Filter	55
Hash	69
Flatten	90
Pipeline	1746
QueryScheduling	1336
Total	**5889**

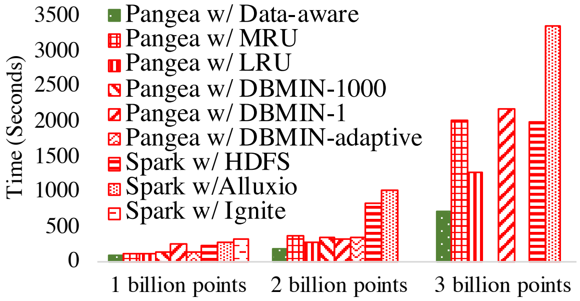


Figure 3: Latency comparison for k-means with five iterations in 11-node cluster using 1-3 billion 10-dimension points (failed cases are shown as gaps).

9.1.1 k-means Clustering

We develop a k-means implementation on Pangea that is similar to the Spark MLlib implementation. We use double arrays to represent points for Pangea and wrap a double array in a Hadoop object as binary input for Spark to minimize (de)serialization overhead. Each run starts with an initialization step that computes norms for each point and samples initial centroids. This is followed by five iterations of the computation.

For Pangea, we use one `write-through` locality set to store input data, and use one `write-back` locality set to store the points with norms for fast distance computation. For Spark, both datasets are cached in memory as RDDs.

Spark runs in Yarn client mode and is tested in two different configurations: Spark over HDFS, Spark using Alluxio as in-memory storage, and Spark using the Ignite SharedRDD. For each test, we tune Spark memory allocations for the Spark executor and the OS, Alluxio, or Ignite for optimal performance. For other parameters, we use the default values. Both Spark and Pangea use 256MB as split/page size.

As shown in Fig. 3, Pangea facilitates k-means processing at up to a 6× speedup compared with Spark. It appears that Pangea's monolithic design facilitates performance gains in several ways:

(1) Reduction in interfacing overhead, including overhead for disk loading, (de)serialization, memory (de)allocation, memory compaction for fragmentation, and memory copies. In Pangea, user data is directly written to buffer pool pages, so when a dataset is imported, a significant portion of it is already cached in the buffer pool without any additional overhead. In Spark, if an external cache like Alluxio or Ignite is not being used, data cannot be shared across applications, which means user data has to be loaded from disk for the initialization step. We find that processing 1 billion points, the initialization step in Spark over HDFS takes 146 seconds, and each of the following iterations only takes 14 seconds; while in Pangea, for processing the same amount of points, the initialization step only takes 43 seconds, and each iteration takes 11 seconds. This shows how much more efficient Pangea is in moving the data to the application the first time. Based on profiling results, Spark over Ignite spends about 40% of time in memory compaction due to fragmentation. De-fragmentation occurs because Ignite seems to be primarily optimized for frequent random access and updates on mutable data, and it enforces a 16KB hard page size limitation. In addition, both Spark over Ignite and Alluxio spend a significant portion of time in object deserialization.

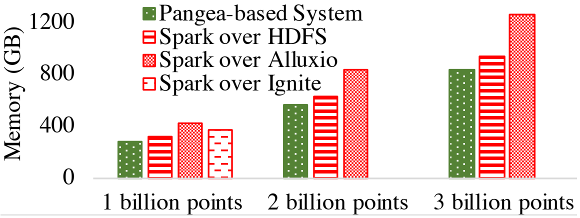


Figure 4: Memory usage (failed cases are shown as gaps).

(2) Removal of redundant data placement. Although using Alluxio as in-memory storage can avoid data loading overhead, double-caching wastes memory resources. Therefore, we observe that when processing 1 billion points using Spark over Alluxio, the initialization step time is 96 seconds, which is 1.5× faster than Spark over HDFS. However, the average latency of following iterations is 37 seconds, which is 3× slower than Spark over HDFS due to the fact that Spark has less working memory (we allocated 15GB memory to Alluxio). Fig. 4 illustrates the memory required by the various setups. The Pangea shared memory pool is configured to be 50GB on each worker node. The total of Spark executor memory and Alluxio worker memory is also limited to 50GB. Ignite requires configuring at least two additional memory sizes: the heap size for each Ignite process (we use 5GB on each node), and the maximum size of the off-heap memory region (we set to 30GB on each node for one billion points). Ignite throws a segmentation fault when processing 2 billion or more points.

(3) Better paging strategy. We compare various paging strategies in Fig. 3. We implement three DBMIN algorithms using three size estimation strategies. DBMIN-adaptive estimates locality set size exactly following the algorithm in [21], while the reference patterns are learned from Pangea-provided services. For DBMIN-1, all locality set sizes are estimated as 1 page. DBMIN-1000 always estimates the size as 1000 pages. Note that DBMIN blocks when the total desired size of all locality sets exceeds available size, which is the reason for the failures of DBMIN-adaptive and DBMIN-1000, as shown in Fig. 3. We find that the data-aware paging strategy significantly outperforms other paging strategies. As mentioned in the implementation of k-means on both platforms, input data needs to be first transformed into a new dataset that has norms associated, which increases the size of working set. Thus, paging is required at 2 billion points.

9.1.2 TPC-H

Our Pangea-based relational query processor and API required around 6,000 lines of C++ code to implement eleven different modules, as illustrated in Tab. 2. While 6,000 lines may seem like a fairly substantial effort, we have, in effect, implemented a high-performance distributed query processing engine with that effort. In addition, we developed around 600 lines of shell and python scripts code for installing and running this computation framework.

We then implement nine different TPC-H benchmark queries on top of our analytics engine. Most of those queries involve aggregation and join. We compare our implementation with an open source, third-party TPC-H implementation[3], which uses Scala and the Spark DataFrame API.

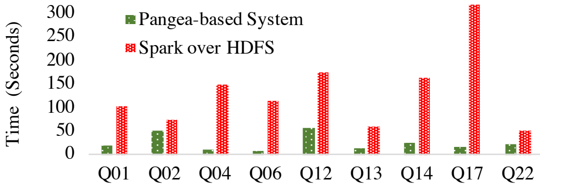


Figure 5: Spark vs. Pangea latency (unit: second) for TPC-H queries in 11-node cluster using scale-100 data.

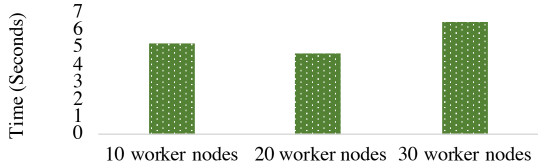


Figure 6: Recovery latency (unit: second) for TPC-H scale-100 data in clusters with 10 to 30 worker nodes.

In Pangea, the `lineitem` source set is a randomly dispatched set which has two replicas, partitioned by `l_orderkey` and `l_partkey` respectively; and the `order` set also has two replicas, partitioned by `o_orderkey` and `o_custkey` respectively. Among the nine TPC-H queries, `Q04` and `Q12` are running on the `lineitem` set that is partitioned by `l_orderkey` and the `order` set that is partitioned by `o_orderkey`; `Q13` and `Q22` are running on the `order` set that is partitioned by `o_custkey`; `Q14` and `Q17` are running on the `lineitem` set that is partitioned by `l_partkey`; and all other queries run on the source sets.

We generate 100GB of TPC-H data (Scale-100), then compare the Pangea-based system with Spark over HDFS, and the results are shown in Fig. 5. By using the heterogeneous replicas for the same table, Pangea applications can achieve up to 20× speedup compared with Spark using the DataFrame API. Note that there is nothing analogous to pre-partitioning available to a Spark developer when loading data from HDFS; all partitioning must be performed at query time. Although a Spark developer could materialize the repartitioned dataset as a different HDFS or Alluxio file, the process of loading data into RDD cache is controlled by Spark task scheduler, which doesn't guarantee locality, thus repartition at runtime is still needed to perform a local join. In addition, such manual replicas can not be utilized by HDFS for failure recovery.

Among those queries, Q17 can achieve 20× speedup, mainly because by selecting the replica of the `lineitem` set that is partitioned by `l_partkey`, and the replica of the `part` set that is partitioned by `p_partkey`, the inputs for the large-scale join in Q17 are co-partitioned, the query scheduler recognizes this by comparing the available partition schemes of both sets through the `statistics` service also provided by Pangea, and pipelines the join operation at each worker node without need to do a repartition.

Failure Recovery. As shown in Fig. 6, for the single-node failure case, recovering the `lineitem` table (with 79GB of raw data) in a ten-node cluster using Pangea's heterogeneous replication only takes five seconds' time, with less than 9% of objects conflicting. The ratio of conflicting objects declines significantly with the increase in number of working nodes: 3% for 20 worker nodes, and zero for 30 worker nodes, as described in Sec. 7.

These results illustrate that Pangea's heterogeneous replication scheme is effective. Although using multiple replicas increase storage size, it is a known and widely accepted cost for high availability.

9.2 Evaluation of Pangea Services

In this subsection, we provide some mico-benchmarks of the various Pangea services.

9.2.1 Sequential Read/Write

This micro-benchmark consists of two tests: one for transient data and one for persistent data. For both, we first write a varying number of 80-byte character array objects to different storage locations, and then we scan those objects; for each object we compute the sum of all the bytes. We run the scanning process repeatedly for five times. In the end, we delete all data.

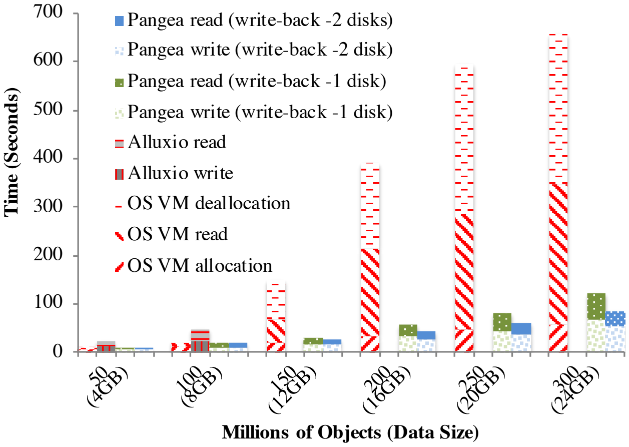


Figure 7: Sequential access for transient data.

Transient Data Test. For this test, we compare Pangea with OS virtual memory (abbrev. OS VM) and Alluxio. For OS VM, we use `malloc()` and `free()` for allocation/deallocation. For Alluxio, we configure the worker memory size to be 14GB and develop a Java client that uses a NIO `ByteBuffer` to efficiently write data to Alluxio worker, which facilitates a 3× speedup compared with using a JNI-based C++ client.

For Pangea, we use a write-back locality set as the data container, so that a write request will return immediately once data is created in the buffer pool.

Results are illustrated in Fig. 7. We observe that when the working set fits in available memory ($<$ 150 millions of objects in our case), the performance of Pangea and OS VM are similar, and both are significantly better than the Alluxio in-memory file system, presumably because using Alluxio requires significant interfacing overhead.

When the working set size exceeds available memory, Pangea can achieve a 5.4× to 7× speedup compared with OS VM, mainly because Pangea increases I/O throughput by using 64MB buffer pool page size and also reduces I/O volume through better paging decisions. Specifically, there is only one locality set, for which Pangea automatically chooses the MRU policy for its sequential access pattern. The OS VM uses an LRU policy and other complex techniques such as page stealing that will evict pages even when

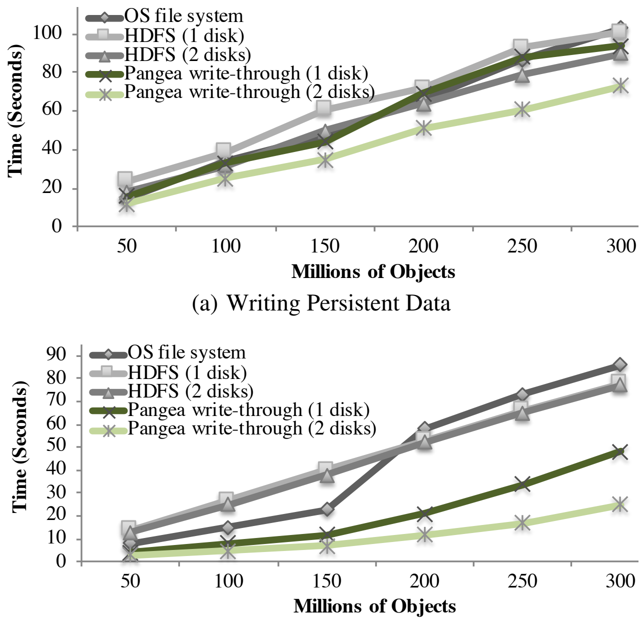
[3] https://github.com/ssavvides/tpch-spark

there is no paging demand. In the case of scanning 200 million objects, for each iteration, the Pangea cache will incur 31.4 page-out operations with 2009.6MB data written to disk on average. However when relying on OS VM, by aggregating the page-out rate collected from the `sar -B` command contained in linux `sysstat` utilities, we see that for each scan iteration, the average size of data written to disk by page-out operations is 5074.2MB (2.5× of Pangea).

We also observe that using Pangea 64MB page has a 2.45× speedup compared with using OS VM 4KB page for writing.

Alluxio doesn't support writing more data than its configured memory size.

Finally, both Pangea and Alluxio are very efficient at removing data. Because data are organized in large blocks in memory, we can deallocate all data belonging to the same block at once. This circumvents the cost of individual object deallocation which accounts for significant overhead, even in C++ applications.



(a) Writing Persistent Data

(b) Reading Persistent Data

Figure 8: Sequential access for persistent data

Persistent Data Test. For this test, we use a write-through locality set for Pangea, so that each page will be persisted to disk via direct I/O immediately when it is fully written. We compare Pangea with the OS filesystem and HDFS. For HDFS, we use the native C++ client developed by Cloudera called `libhdfs3` to avoid JNI overhead and provide a fair comparison.

The results are illustrated in Fig. 8. After careful tuning, the writing performance of the three systems is similar. However, for average latency of one iteration of scan with simple computation, Pangea outperforms the OS filesystem by a factor of 1.9× to 2.7× and outperforms HDFS by 1.5× to 3.5×.

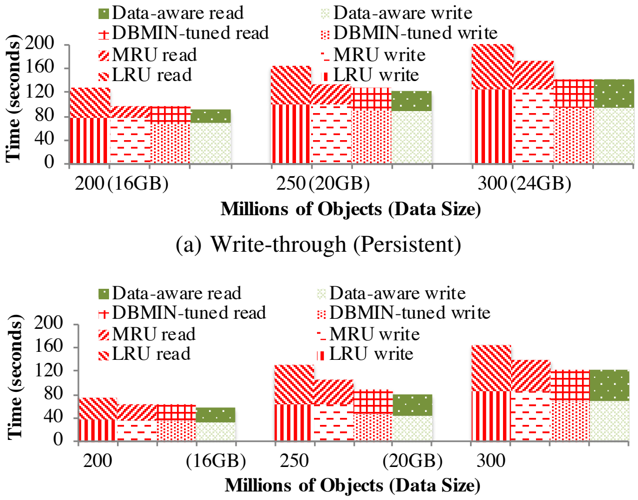
Through profiling, we find the performance gain when the working set fits in memory is mainly from the reduction in interfacing overhead; the Pangea client writes to shared memory directly, and those writes are flushed to the file system directly. Thus, we can avoid the memory copy overhead between user space and kernel space as required by the OS buffer cache and also avoid the memory copy between client and server as required by HDFS.

When the working set size exceeds available memory size, I/O becomes the performance bottleneck, and the Pangea data-aware paging strategy can significantly reduce page swapping, which is the root cause of the performance gain in this case.

Paging Strategy Comparison. The data-aware paging strategy adopts MRU for sequential access pattern. The most recently used unpinned page will be evicted from a victim locality set under writing, and at most 10% of the most recently used unpinned pages will be evicted from a victim locality set under reading.

We compare above strategy with MRU, LRU, and DBMIN for sequential access. In our implementation, 10% of the most recently used pages will be evicted at each eviction for MRU, and at most 10% of the least recently used pages will be evicted for LRU. Compared with OS VM paging, Pangea does not use page stealing, which means it will not evict pages when there is no paging demand. There is only one locality set and it is repeatedly read after being written. For such a loop-sequential pattern, the original DBMIN algorithm suggests configuring the size of the locality set to be the set size. For a set exceeding memory size, DBMIN will block. To avoid this, we upper-bound the locality set size at the memory size.

Fig. 9 lists the comparison results for using one disk. The results for using two disks are similar. For reading, the Pangea data-aware policy, tuned DBMIN policy and MRU can achieve 1.6× to 2.5× speedup compared with LRU. This is because in such a read-after-write scenario—which is common in dataflow processing—LRU tends to evict pages that will be read immediately after being evicted.



(a) Write-through (Persistent)

(b) Write-back (Transient)

Figure 9: Page replacement for sequential access.

The Pangea data-aware policy can achieve up to 50% performance improvement compared with LRU and MRU, and up to 20% compared with the tuned DBMIN algorithm. This is mainly because with the knowledge about on-going operations (i.e. read or write), Pangea can reduce the number of dirty pages to evict to minimize expensive disk writing operations as compared with LRU and MRU; and can evict more unused pages each time for reading, to better overlap I/O operations with computation as compared to DBMIN.

From Fig. 9, we see that reading write-back data is slower than the write-through data. That is because for the former case, transient pages may be written back to disk in the reading phase, while for the latter case, all pages are flushed to disk in the writing phase.

9.2.2 *Shuffle*

For this micro-benchmark, to provide an apples-to-apples comparison (not JVM vs. C), we compare Pangea's shuffle service to simulated Spark shuffling written in C++. In Spark shuffling, each CPU core will have a separate spill file in the local file system for each shuffle partition, so there will be $numCores \times numPartitions$ files in total. For Pangea, all data belonging to the same partition are written to one locality set, so there are at most $numPartitions$ spill files.

In our test setup, each worker generates small strings of about 10 bytes in length. For each string, a worker computes its partition via a hash function. For reading shuffle data, each worker reads all strings belonging to one partition, and for each string, the worker scans each byte and adds up the byte value.

We use four workers to write to four partitions and four workers to read from the four partitions. The performance results are illustrated in Table. 3, which show that we can achieve 1.1× - 1.4× speedup for shuffle writing and 2.2× - 27× speedup for shuffle reading.

When the working set fits in memory (when the per-thread data size is smaller than 3500 MB), the performance gain of Pangea is mainly from the reduction of memory allocation and copy overhead, because in Pangea, the objects for shuffling are directly allocated in a small page using a sequential allocator. However, for Spark shuffling, data needs to be first allocated on heap (we use `malloc()` for the C++ implementation) and then written to file (we use `fwrite()` for the C++ implementation).

When the working set size exceeds memory (the per-thread data size is larger than 3500 MB), the performance gain of Pangea is mainly from significant reduction in I/O overhead that is brought by a smaller number of files and better page replacement decisions.

Table 3: Shuffle data read and write latency with 4 writing/reading workers (unit: seconds).

MB/thread	C Spark shuffle		Pangea (1 disk)		Pangea (2 disks)	
	write	read	write	read	write	read
500	21	**5**	15	**<1**	15	<1
1000	43	**9**	31	**<1**	31	<1
1500	65	**13**	47	**<1**	47	<1
2000	86	**19**	63	**<1**	63	<1
2500	107	**23**	80	**<1**	79	1
3000	129	**27**	94	**<1**	94	1
3500	152	**36**	115	**10**	114	6
4000	172	**47**	140	**15**	134	12
4500	194	**55**	157	**20**	155	14
5000	215	**64**	189	**27**	174	17
5500	237	**70**	216	**32**	196	26
6000	259	**78**	235	**35**	215	32

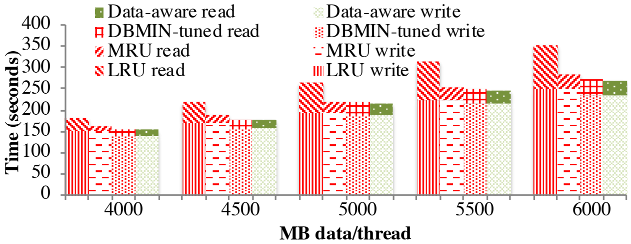


Figure 10: Page replacement comparison for shuffle.

Fig. 10 shows the comparison results of different paging policy for shuffle with one disk. The data-aware paging policy outperforms LRU in reading by up to 3×. That is because in the reading process, when using a data-aware policy, the first 223 pages are kept in the buffer pool without being flushed and can be directly read by reading workers. Thus the number of I/O operations is observed to be significantly reduced. The Pangea data-aware policy also outperforms LRU and MRU in writing speed by around 10%, and outperforms the tuned DBMIN policy in reading speed by around 10%; that is because by using the former policy, we can distinguish on-going operations (read or write) through the locality set's `CurrentOperation` attribute and can optimize the number of pages to evict for different operations.

9.2.3 *Hash Aggregation*

We aggregate varying numbers of randomly generated `<string, int>` pairs, following the incise.org benchmark [4]. We compare Pangea with an STL unordered_map and Redis 3.2.0, which is a well-known high performance key-value store developed in C++ [50]. The results are illustrated in Tab. 4. We see that Pangea hashmap outperforms STL unordered_map by up to 50×, and outperforms Redis by up to 30×.

The Pangea hashmap is initialized to have 200 partitions. The STL unordered_map starts to swap virtual memory when inserting 200 million keys; however the Pangea hashmap starts spilling to disk only when inserting 300 million keys. That is mainly because the memcached slab allocator that is used as a secondary data allocator in Pangea has better memory utilization than the STL default allocator. Redis incurs significant latency because it adopts a client/server architecture, and is inefficient for problems where the computation can directly run on local data.

Table 4: Key-value pair aggregation (unit: seconds).

NumKeys	STL unordered_map	Pangea hashmap	Redis
50,000,000	47	**33**	53
100,000,000	38	**68**	274
150,000,000	153	**110**	2069
200,000,000	7657	**167**	9103
250,000,000	16818	**332**	9887
300,000,000	>7 hours	**2450**	failed

Summary. The experiments show that using Pangea services can bring up to a 6× speedup for k-means, and up to 20× speedup for TPC-H. In addition, various micro-benchmarks demonstrate that Pangea can provide high-performance sequential scan, sequential write, shuffle, hash aggregation services, which are all important building blocks for modern analytics.

10. CONCLUSIONS

There are multiple layers in modern data analytics systems for managing shared persistent data, cached data, and non-shared execution data. These layers are implemented by separate systems such as HDFS, Alluxio, Ignite, and Spark. Such layering introduces significant performance and management costs. Pangea is designed and implemented to solve this problem through the locality set abstraction at a single layer. A locality set can be aware of application semantics through services provided within Pangea, and use this information for data placement, page eviction and so on. The results show that Pangea is a promising alternative base for building performance-critical applications and distributed data analytics tools.

11. ACKNOWLEDGEMENTS

Thanks to Joseph Hwang for implementing the first prototype of the manager communication component. Thanks to Donpaul Stephens, Andrea C. Arpaci-Dusseau, and anonymous reviewers for their insightful feedbacks on earlier versions of this paper. Work presented in this paper has been supported by the DARPA MUSE program, award No. FA8750-14-2-0270 and by the NSF under grant Nos. 1355998 and 1409543.

12. REFERENCES

[1] Amazon simple storage system. https://aws.amazon.com/s3.

[2] Apache ignite. https://ignite.apache.org.

[3] Google cloud storage. https://cloud.google.com/storage.

[4] Hash table benchmark. http://incise.org/hash-table-benchmarks.html.

[5] Project tungsten: Bringing spark closer to bare metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[6] Why enterprises of different sizes are adopting 'fast data' with apache spark. https://www.lightbend.com/blog/why-enterprises-of-different-sizes-are-adopting-fast-data-with-apache-spark.

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow. org*.

[8] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370. ACM, 2004.

[9] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The stratosphere platform for big data analytics. *The International Journal on Very Large Data Bases*, 23(6):939–964, 2014.

[10] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 20–20. USENIX Association, 2012.

[11] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[12] J. Arnold. *Openstack swift: Using, administering, and developing for swift object storage*. " O'Reilly Media, Inc.", 2014.

[13] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in the batch-aware distributed file system. In *NSDI*, volume 4, pages 365–378, 2004.

[14] D. Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[15] D. P. Bovet and M. Cesati. *Understanding the Linux kernel*. " O'Reilly Media, Inc.", 2005.

[16] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[17] P. Cao and et al. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *TOCS*, 14(4):311–343, 1996.

[18] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems*, volume 12, pages 193–206, 1997.

[19] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[20] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *PVLDB*, 5(12):1802–1813, 2012.

[21] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, 1(1-4):311–336, 1986.

[22] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. An architecture for compiling udf-centric workflows. *PVLDB*, 8(12):1466–1477, 2015.

[23] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1-2):515–529, 2010.

[24] D. Ellard, E. Thereska, G. R. Ganger, M. I. Seltzer, et al. Attribute-based prediction of file properties. 2003.

[25] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. CoHadoop: flexible data placement and its exploitation in hadoop. *PVLDB*, 4(9):575–585, 2011.

[26] R. Fagin and T. G. Price. Efficient calculation of expected miss ratios in the independent reference model. *SIAM Journal on Computing*, 7(3):288–297, 1978.

[27] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[28] R. Fonseca, V. Almeida, M. Crovella, and B. Abrahao. On the intrinsic locality properties of web reference streams. Technical report, Boston University Computer Science Department, 2002.

[29] M. Garetto, E. Leonardi, and S. Traverso. Efficient analysis of caching strategies under dynamic content popularity. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 2263–2271. IEEE, 2015.

[30] S. Ghemawat and et al. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[31] K. Gupta and et al. GPFS-SNC: An enterprise storage framework for virtual-machine clouds. *IBM Journal of Research and Development*, 55(6):2–1, 2011.

[32] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010.

[33] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation reuse in analytics job service at microsoft. In *Proceedings of the 2018 International Conference on Management of Data*, pages 191–203. ACM, 2018.

[34] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, pages 117–134, 2016.

[35] L. Kleinrock. *Queueing systems, volume 2: Computer applications*, volume 66. Wiley New York, 1976.

[36] M. Kornacker and J. Erickson. Cloudera Impala: Real time queries in apache hadoop, for real. *ht tp://blog. cloudera. com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real*, 2012.

[37] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE*

transactions on Computers, (12):1352–1361, 2001.

[38] H. Li. Alluxio: A virtual distributed file system. 2018.

[39] H. Li and et al. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SOCC*, pages 1–15, 2014.

[40] J. Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[41] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng. Lifetime-based memory management for distributed data processing systems. *PVLDB*, 9(12):936–947, 2016.

[42] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A new dynamic memory allocator for real-time systems. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 79–88. IEEE, 2004.

[43] M. Mesnier, E. Thereska, G. R. Ganger, D. Ellard, and M. Seltzer. File classification in self-* storage systems. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 44–51. IEEE, 2004.

[44] A. Morton. Usermode pagecache control: fadvise ().

[45] R. Nishtala and et al. Scaling memcache at facebook. In *NSDI*, pages 385–398, 2013.

[46] E. J. O'neil and et al. The lru-k page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, 1993.

[47] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified i/o buffering and caching system. *ACM Transactions on Computer Systems (TOCS)*, 18(1):37–66, 2000.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[49] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 558–569. ACM, 2002.

[50] S. Sanfilippo and P. Noordhuis. Redis, 2009.

[51] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. *PVLDB*, 6(5):325–336, 2013.

[52] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[53] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[54] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2012.

[55] M.-J. Wu, M. Zhao, and D. Yeung. Studying multicore processor scaling via reuse distance analysis. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 499–510. ACM, 2013.

[56] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.

[57] M. Zaharia and et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–15. USENIX, 2012.

[58] J. Zhou, N. Bruno, and W. Lin. Advanced partitioning techniques for massively distributed computation. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 13–24. ACM, 2012.

[59] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference, General Track*, pages 91–104, 2001.

[60] J. Zou, R. M. Barnett, T. Lorido-Botran, S. Luo, C. Monroy, S. Sikdar, K. Teymourian, B. Yuan, and C. Jermaine. PlinyCompute: A platform for high-performance, distributed, data-intensive tool development. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1189–1204. ACM, 2018.