Chapter 4

DATA MANAGEMENT TECHNIQUES

Angelos Bilas[1] and Jesus Carretero[2] and Toni Cortes[3] and Javier Garcia-Blas[4] and Pilar González-Férez[5] and Anastasios Papagiannis[6] and Anna Queralt[7] and Fabrizio Marozzo[8] and Giorgos Saloustros[9] and Ali Shoker[10] and Domenico Talia[11] and Paolo Trunfio[12]

Keywords: Availability, Scalability, CRDT, DataClay, Key-value stores, SSDs, memory-mapped I/O, RDMA-based communication, Data-centric, Data-locality, Workflows, Cloud Computing, DMCF, Hercules

Today, it is projected that data storage and management is becoming one of the key challenges in order to achieve Ultrascale computing for several reasons. First, data is expected to grow exponentially in the coming years and this progression will imply that disruptive technologies will be needed to store large amounts of data and more importantly to access it in a timely manner. Second, the improvement of computing elements and their scalability are shifting application execution from CPU bound to I/O bound. This creates additional challenges for significantly improving the access to data to keep with computation time, and thus avoid HPC from being underutilized due to large periods of I/O activity. Third, the two initially separate worlds of HPC that mainly consisted on one hand of simulations that are CPU bound and on the other hand of analytics that mainly perform huge data scans to discover

[1]Institute of Computer Science, FORTH (ICS), Heraklion, Greece
[2]ARCOS, University Carlos III, Madrid, Spain
[3]BSC and Universitat Politécnica de Catalunya, Spain
[4]ARCOS, University Carlos III, Madrid, Spain
[5]Department of Computer Engineering, University of Murcia, Spain
[6]Institute of Computer Science, FORTH (ICS), Heraklion, Greece
[7]Barcelona Supercomputing Center (BSC), Spain
[8]DIMES, University of Calabria, Rende, Italy
[9]Institute of Computer Science, FORTH (ICS), Heraklion, Greece
[10]HASLab, INESC TEC & University of Minho, Portugal
[11]DIMES, University of Calabria, Rende, Italy
[12]DIMES, University of Calabria, Rende, Italy

information and are I/O bound, is blurring. Now, simulations and analytics need to work cooperatively and share the same I/O infrastructure.

This challenge of data management is currently being addressed from many different angles by a large international community, but there are three main concepts in which there is a wide agreement:

- First, data abstractions need to change given that traditional parallel file systems are failing on adapting to the new needs of applications and especially to their scalability characteristics. For this reason, abstractions such as object stores or key-values are positioning themselves in the lead to become the new storage containers. Examples such as Dynamo, BigTable or Cassandra, among others, are being used by companies like Facebook or Twitter that need to manage and process incredibly huge amounts of data.
- Second, data needs to be replicated in order to both become resilient to errors in the vast amount of hardware needed to store it and to offer fast enough access to it. Unfortunately, this replication implies significant overheads in both the space needed and the access/modification time of the data, which were the original challenges.
- Third, the use of HPC is becoming more necessary than ever in order to be able to analyze the huge amount of data that is generated everywhere (i.e. smart cities, cars, open data, etc.), thus traditional HPC systems cannot be the only ones used to analyze this data because they are scarce and very expensive to use. For this reason, cloud environments are becoming closer to HPC in performance and cheaper to use taking an active role in solving HPC problems of standard users.

In this chapter we will present three techniques that address the aforementioned challenges. First, we will present Tucana, a novel key-value store designed to work with fast storage devices such as SSDs. In addition to presenting its rich-features including arbitrary dataset sizes, variable key and value sizes, concurrency, multi-threading, and versioning, we will evaluate its performance compared with similar key-value components with respect to throughput, CPU usage, and IO. This novel key-value system advances in the idea that new abstractions to store data are needed in Ultra Scale computing.

Second, we present Hercules and the benefits of integrating it and a Data Mining Cloud Framework as an alternative to classical parallel file system enabling the execution of data intensive application on cloud environments. Furthermore, we detail a data-aware scheduling technique that can reduce the execution time of data intensive workflows significantly. This platform combination, in addition to the new data scheduler, puts a step forward in the use of cloud infrastructures for Ultra scale computing by a larger community whose needs are growing.

Finally, this chapter discuss the idea of Conflict-free Replicated Data Types (CRDT), a methodology that helps implementing basic types that can be replicated without the overhead of synchronization on the critical path of data update, but with the guarantee that, at some point in time, all replicas will see unequivocally all updated done in the other replicas. Furthermore, basic CRDT can be combined to construct larger datatypes with the same properties. This methodology enables the proliferation of replicas without the overhead of accessing/modifying them which will be key in Ultra Scale computing.

4.1 Intra-node Scaling of an Efficient Key-value Store on Modern Multicore Servers

Tucana is a key-value store designed for fast storage devices, such as SSDs, that reduces the gap between sequential and random I/O performance, especially under high degree of concurrency and relatively large I/Os (a few tens of KB). It supports variable size keys and values, versions, arbitrary data set sizes, concurrency, multithreading, and persistence. *Tucana* uses a B$^{\varepsilon}$–tree approach that only performs buffering and batching at the lowest part of the tree, since it is assumed that the largest part of the tree (but not the data items) fits in memory. Comparing to RosckDB, *Tucana* is up to $9.2\times$ more efficient in terms of CPU cycles/op for in-memory workloads and up to $7\times$ for workloads that do not fit in memory.

This section analyzes in detail host CPU overhead, network and I/O traffic of H-*Tucana*, a modification of HBase that replaces the LSM-based store engine of HBase with *Tucana*. Several limitations to achieve higher performance are identified. First, network path, inherited from HBase, becomes a bottleneck and does not allow *Tucana* to saturate server cores. Second, lookups for traversing the tree represent up to 78% of the time used by *Tucana*, mainly due to key comparisons. Finally, when mmap, used by *Tucana* for allocating memory and device space, runs out of memory, *Tucana* exhibits periods of inactivity and stops serving requests, until the system replenishes available buffers for mmap.

4.1.1 Introduction

Currently, key-value (KV) stores have became an important building block in data analytics stacks and data access in general. Today's large-scale, high-performance data-intensive applications usually use KV stores for data management, since they offer higher efficiency, scalability, and availability than relational database systems. KV stores are widely used to support Internet services, for instance, Amazon uses Dynamo, Google uses BigTable, Facebook and Twitter use Cassandra and HBase.

The core of a NoSQL store is a key-value store that performs (key,value) pair lookups. Traditionally key-value stores have been designed for optimizing accesses to hard disk drives (HDDs) and with the assumption that the CPU is the fastest component of the system (compared to storage and network devices). Indeed, key-value stores tend to exhibit high CPU overheads [166].

Tucana [166] is a key-value store designed for fast storage devices, such as SSDs, that reduces the gap between sequential and random I/O performance, especially under high degree of concurrency and relatively large I/Os (a few tens of KB). *Tucana* is a full-feature key-value store that achieves lower host CPU overhead per operation than other state-of-the-art systems. It supports variable size keys and values, versions, arbitrary data set sizes, concurrency, multithreading, and persistence. The design of

Tucana centers around three techniques to reduce overheads: copy-on-write, private subsection allocation, and direct device management.

Tucana is up to $9.2\times$ more efficient in terms of CPU cycles/op for in-memory workloads and up to $7\times$ for workloads that do not fit in memory [166]. *Tucana* outperforms RocksDB for in memory workloads up to $7\times$, whereas for workloads that do not fit in memory both systems are limited by device I/O throughput.

Tucana is used to improve the throughput and efficiency of HBase [167], a popular scale-out NoSQL store. The LSM-based storage engine of HBase is replaced with *Tucana*. Data lookup, insert, delete, scan, and key-range split and merge operations are served from *Tucana*, while maintaining the HBase mapping of tables to key-value pairs, client API, client-server protocol, and management operations (failure handling and load balancing). The resulting system, called H-*Tucana*, remains compatible with other components of the Hadoop ecosystem. H-*Tucana* is compared to HBase using YCSB and results show that, compared to HBase, H-*Tucana* achieves between $2-8\times$ better CPU cycles/op and $2-10\times$ higher operation rates across all workloads.

This section uses *Tucana* to study the behavior of H-*Tucana* by using two datasets, one that fits in memory and one that does not. In addition, H-*Tucana*'s behavior is compared with HBase and with two ideal versions of H-*Tucana* that do not perform I/O. The aim is to understand the overheads associated with efficient key value stores for fast storage devices.

The aspect examined are: host CPU overhead, network and I/O traffic, the use of RAM, and write amplification problem, i.e., whether the same data is written multiple times. Results show that there are several important limitations to achieve higher performance. Although the average CPU utilization is low, a few (one or two) cores, that execute tasks of the network path, become bottleneck. The analysis for network traffic shows that network devices are far from achieving their maximum throughput. Therefore, the network path, inherited from HBase, does not allow *Tucana* to achieve better performance. Regarding index metadata, traversing the *Tucana* tree represents up to 78% of the time used by *Tucana* mainly due to the key comparisons. In addition, when mmap runs out of memory, the key-value store exhibits periods of inactivity and stops serving requests, until the system replenishes available buffers for mmap. Next, these issues are analyzed and evaluated in more detail.

4.1.2 Background

KV stores have been traditionally designed for HDDs and use LSM-tree structure at their core. LSM-trees [168] are a write-optimized structure that is a good fit for HDDs where there is a large difference in performance between random and sequential accesses. LSM-trees organize data in multiple levels of large, sorted containers, where each level increases the size of the container.

Their advantages are: (1) they require a small amount of metadata, since data containers are sorted; (2) I/Os can be large, resulting in optimal HDD performance. The drawback is that for keeping large sorted containers they perform compactions which incurs high CPU overhead and results in I/O amplification for reads and writes.

Going forward device performance and CPU-power trends dictate different designs. *Tucana* uses as a basis a variant of B-trees, broadly called B^{ε}–trees [169].

B$^{\varepsilon}$–trees are B-trees with an additional per-node buffer. These buffers allow to batch insert operations to amortize their cost. In B$^{\varepsilon}$–trees the total size of each node is B and ε is a design-time constant between [0,1]. ε is the ratio of B that is used for buffering, whereas the rest of the space in each node (1-ε) is used for storing pivots.

Buffers contain messages that describe operations that modify the index (insert, update, delete). These operations are initially added to the tree's root node buffer. When the root node buffer becomes full, a subset of the buffered operations are propagated to the buffers of the appropriate nodes at the next level. This procedure is repeated until operations reach a leaf node, where the key-value pair is simply added to the leaf. Leaf nodes are similar to B-Trees and they do not contain an additional buffer, beyond the space required to store the key-value pairs.

A get operation traverses the path from the root to the corresponding leaf by searching at the buffers of the internal nodes along the path. A range scan is similar to a get, except that messages for the entire range of keys must be checked and applied as the appropriate subtree is traversed. Therefore, buffers are frequently modified and searched. For this reason, they are typically implemented with tree indexes rather than sorted containers.

Compared to LSM-trees, B$^{\varepsilon}$–trees incur less I/O amplification. B$^{\varepsilon}$–trees use an index to remove the need for sorted containers. This results in smaller and more random I/Os. As device technology reduces the I/O size required to achieve high throughput, using a B$^{\varepsilon}$–tree instead of an LSM-tree is a reasonable decision.

4.1.3 Tucana Design

Tucana [166] is a feature-rich key-value store that provides persistence, arbitrary dataset sizes, variable key and value sizes, concurrency, multithreading, and versioning.

Tucana uses a B$^{\varepsilon}$–tree approach to maintain the desired asymptotic properties for inserts, which is important for write-intensive workloads. B$^{\varepsilon}$–trees achieve this amortization by buffering writes at each level of the tree. *Tucana*'s design assumes that the largest part of the tree (but not the data items) fits in memory and *Tucana* only performs buffering and batching at the lowest part of the tree.

Figure 4.1 shows an overview of the tree index organization of *Tucana*. The index consists of internal nodes with pointers to next level nodes and pointers to variable size keys. A separate buffer per internal node stores the variable size keys. Pointers to keys are sorted based on the key, whereas keys are appended to a buffer. The leaf nodes contain sorted pointers to the key-value pairs. A single append-only log is used to store both the key and values. The design of the log is similar to the internal buffers of B$^{\varepsilon}$–trees. However, note that *Tucana* avoids buffering at intermediate nodes. For each key, leaves store as metadata a fixed-size prefix of the key and a hash value of the key, whereas the key itself is stored in the key-value log. These prefix and hash values of the keys will help while traversing the tree.

Insert operations traverse the index in a top down fashion. At each index node, a binary search is performed over the pivots to find the next level node to visit. When the leaf is reached, the key-value pair is appended to the log and the pointer is inserted in the leaf, keeping pointers sorted by the corresponding key. If a leaf is full, a

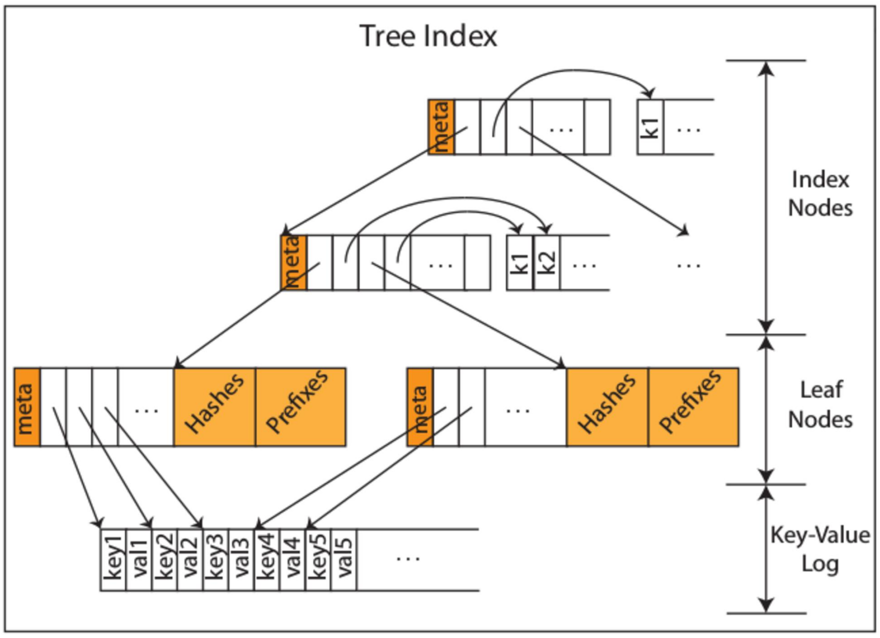


Figure 4.1: Design of the tree index of *Tucana*.

split operation is triggered prior to insert. Split operations, in index or leaf nodes, produce two new nodes each containing half of the keys and they update the index in a bottom-up fashion. Delete operations place a tombstone for the respective keys, which are removed later. Deletes will eventually cause rebalancing and merging.

Point queries locate the appropriate leaf traversing the index similar to inserts. At the leaf, a binary search is performed to locate the pointer to the key-value pair. Finally, range queries locate the starting key similar to point queries and subsequently use the index to iterate over the key range.

Tucana manages the data layout as a set of contiguous segments of space to store data. Each segment is composed of a metadata portion and a data portion. The metadata portion contains the superblock, free log, and segment allocator metadata (bitmap). The superblock contains a reference to a descriptor of the latest persistent and consistent state for a segment. Each segment has a single allocator common for all databases (key ranges) in a segment. The data portion contains multiple databases. Each database is contained within a single segment and uses its own separate indexing structure. *Tucana* keeps persistent state about allocated blocks by using bitmaps.

Tucana directly maps the storage device to memory to reduce space (memory and device) allocation overhead. *Tucana* leverages `mmap` to use a single allocator for memory and device space. `mmap` uses a single address space for both memory and storage and virtual memory protection to determine the location (memory or storage) of an item. This eliminates the need for pointer translation at the expense of page faults. Additionally, `mmap` eliminates data copies between kernel and user space.

Tucana uses copy-on-write (CoW) to achieve recovery without the use of a log. CoW maintains the consistency of both allocator bitmap and tree metadata. The bitmap in each segment consists of *buddy pairs* with information about allocated space. Only one buddy of the pair is active for write operations, whereas the other buddy is immutable for recovery. A global per segment increasing counter, named ***epoch***, marks each buddy. A successful commit operation increments the epoch denoting the latest instant in which the buddy was modified.

For the tree structure, each internal index and leaf node has epochs to distinguish its latest persistent state. During an update, whether the node's epoch indicates that it is immutable, a CoW operation will take place. After a CoW operation for inserting a key, the parent of the node is updated with the new node location in a bottom-up fashion. The resulting node belongs to epoch+1 and will be persisted during the next commit. Subsequent updates to the same node before the next commit are batched by applying them in place. Since keys and values are stored in buffers in an append-only fashion, CoW is only performed on the header of each internal node.

Tucana's persistence relies on the atomic transition between consistent states for each segment. Metadata and data in *Tucana* are written asynchronously to the devices. However, transitions from state to state occur atomically via synchronous updates to the segment's superblock with `msync` (commits). Each commit creates a new persistent state for the segment, identified by a unique epoch id. The epoch of the latest persistent state of a segment is stored in a descriptor to which the superblock keeps a reference.

HBase [167] is a scale-out columnar store that supports a small and volatile schema. HBase offers a table abstraction over the data, where each table keeps a set of key-value pairs. Each table is further decomposed into regions, where each region stores a contiguous segment of the key space. Each region is physically organized as a set of files per column. At its core HBase uses an LSM-tree to store data [168]. *Tucana* is used to replace this storage engine, while maintaining the HBase metadata architecture, node fault tolerance, data distribution and load balancing mechanisms. The resulting system, H-*Tucana*, maps HBase regions to segments, and each column to a separate tree in the segment. To eliminate the need for using HDFS under HBase, HBase is modified so that a new node handles a segment after a failure. It is assumed that segments are allocated over a reliable shared block device, such as a storage area network (SAN) or virtual SAN and are visible to all nodes in the system. In this model, the only function that HDFS offers is space allocation. *Tucana* is designed to manage space directly on top of raw devices, and it does not require a file system. H-*Tucana* assumes the responsibility of elastic data indexing, while the shared storage system provides a reliable (replicated) block-based storage pool.

4.1.4 Experimental Evaluation

The experimental platform consists of two machines each equipped with a 16 core Intel(R) Xeon(R) E5-2630 CPU running at 2.4 GHz and 64 GB DDR4 DRAM. Both nodes are connected with a 40 Gbits/s network link. As storage device, the server uses a Samsung SSD PRO 950 NVMe of 256 GB.

The open-source Yahoo Cloud Serving Benchmark (YCSB) [170] is. used to generate synthetic workloads. The default YCSB implementation executes gets as range queries and therefore, exercises only scan operations. For this reason, YCSB is modified to use point queries for get operations. Two standard workloads proposed by YCSB with the default values are run. Table 4.1 summarizes these workloads.

	Workload
A	50% reads, 50% updates
C	100% reads

Table 4.1 Workloads evaluated with YCSB. All workloads use a query popularity that follows a Zipf distribution.

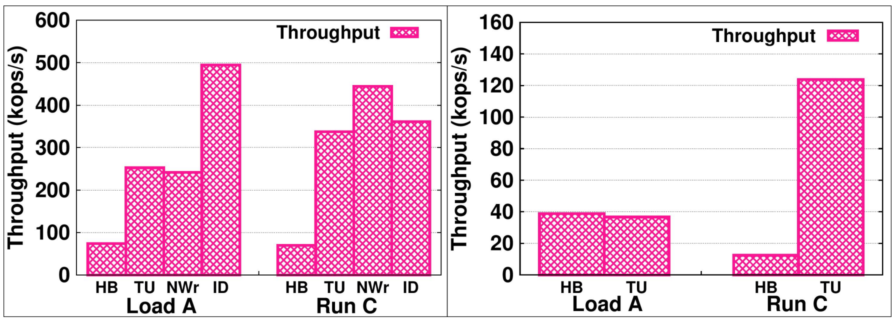
The following sequence is run: Load the database using workload A's configuration file, and then run workload C. In this way, the experiment has two phases an initial one with only puts operations (write requests), and a second phase with only gets operations (read requests). However, due to the commits at the beginning of the run phase The run uses 32 YCSB threads and eight regions. Two datasets are used, a small one that fits in memory and a large one that does not. The small dataset is composed of 100M records, and the large dataset has 500M records. The load phase creates the whole dataset and the run phase issues 10 million operations.

The analysis uses four engines: HBase, H-*Tucana*, NW-H-*Tucana*, and Ideal-H-*Tucana*. H-*Tucana* is cross-linked between the Java code of HBase and the C code of *Tucana*. NW-H-*Tucana* is a modification that does not perform any I/O by completing all the I/O requests without issuing them to the storage device. The *Tucana* tree is used, and the get and put operations are performed, but all the data remains in memory. Ideal-H-*Tucana* is a modification in which *Tucana* completes put requests without doing the insert operation and get requests return a dummy value. The *Tucana* tree is not used and no I/O is performed either. For NW-H-*Tucana* and Ideal-H-*Tucana* only the small dataset is used.

Graphs for CPU utilization depict values given by `mpstat`, and include: (i) User that corresponds to %usr + %nice; (ii) Sys that corresponds to %sys + %irq + %soft, where %irq and %soft correspond to the percentage of time spent by the CPUs to service hardware interrupts and software interrupts, respectively; and (iii) IOW that corresponds to %iowait the percentage of time that the CPUs were idle during which the system had an outstanding disk I/O request.

4.1.4.1 Throughput analysis

Figure 4.2 depicts the throughput, in Kops/s, achieved by HBase and H-*Tucana* with both datasets, and by NW-H-*Tucana* and Ideal-H-*Tucana* with the dataset that fits in memory. Regarding H-*Tucana* and comparing both operations, the run phase (get operations) outperforms the load phase (put operations) by 33.7% and 3.4$\times$ with the small and large datasets, respectively. There are several reasons for this difference in performance. First, *Tucana* uses a lock for inserting new KV pairs, therefore put operations require a lock to ensure exclusive access when modifying the tree. On the contrary, *Tucana* provides lock-less gets. Second, the amount of I/O traffic during the load phase is significantly larger than during the run phase. During the load phase, I/O write operations are issued to ensure that the dataset is stored on the device, whereas during the run phase there are almost not writes with the exception of a snapshot at the beginning of the phase. In addition, for the small dataset, during the run phase there



(a) 100M records (b) 500M records

Figure 4.2: Throughput (kops/s) achieved by HBase, H-*Tucana*, and the NW and Ideal versions of H-*Tucana* with the 100M dataset, and by HBase and H-*Tucana* with 500M records.

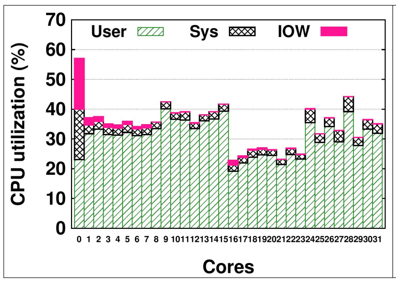
are no reads since the dataset fits in memory, and for the large dataset, the amount of data read is smaller than during the load phase. Finally, for the large dataset, the memory pressure has a huge impact during the load phase, since index and leaf nodes are evicted from memory and they have to be re-read several times, and they are written multiple times.

Comparing performance to HBase, with the small dataset, H-*Tucana* significantly outperforms HBase by up to 3.4× and 4.8× during the load and run phases, respectively. Thanks to its B$^{\varepsilon}$–tree approach, when dataset fits in memory H-*Tucana* is able to significantly improve HBase performance. However, with the large dataset, H-*Tucana* exhibits up to 5.7% worst performance than HBase during the load phase, whereas H-*Tucana* outperforms HBase by up to 9.9× during the run phase.

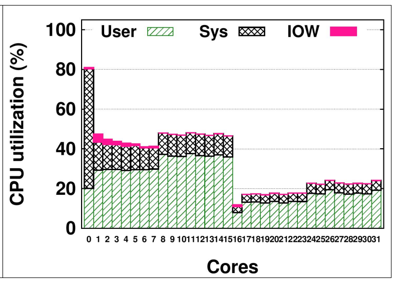
HBase stores data on the device as HFiles, and the dataset is only sorted per HFile. New insert operations do not need to read previous values. HBase usually issues large write operations achieving good I/O performance. The size of the dataset does not significantly impact on its write performance. However, for get operations, this organization exhibits poor read performance when the dataset does not fit in memory, since all HFiles have to be checked for each get. HBase (and LSM trees in general) mitigate these overheads with the use of compactions and bloom filters.

With the large dataset and during the load phase, the problem of H-*Tucana*, as subsection 4.1.4.3 shows, is the significant amount of data written, and the bad I/O pattern produced. This problem is not inherent to the design of *Tucana* tree, but rather due to `mmap`. subsection 4.1.4.3 shows that H-*Tucana* writes less amount of data than HBase, but it issues smaller requests that ends in worst I/O performance.

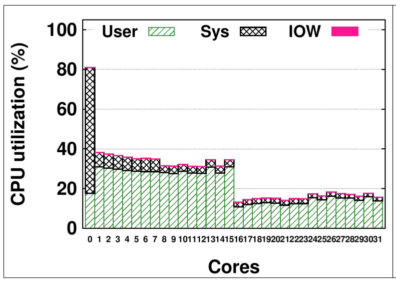
During the load phase, NW-H-*Tucana* provides quite similar performance than H-*Tucana*, the reason is that for small datasets and fast devices, the role of the device is reduced and the differences between NW-H-*Tucana* and H-*Tucana* are small. However, during the run phase, NW-H-*Tucana* improves H-*Tucana* performance by up to 32%. This difference in performance is due to the snapshot performed by H-*Tucana* at the beginning of the tests that issues I/O traffic and reduces H-*Tucana* performance. When comparing both phases, get operations outperform by up to 84%



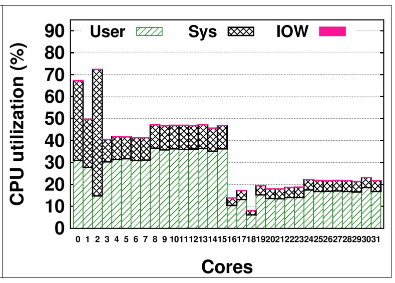
(a) HBase



(b) H-*Tucana*

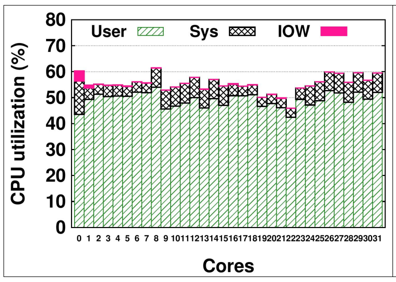


(c) NW-H-*Tucana*

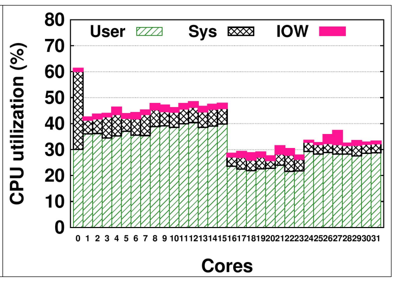


(d) Ideal-H-*Tucana*

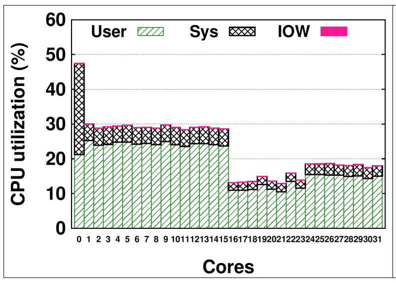
Figure 4.3: CPU utilization per core at the server side during the execution of the load phase with 100M records.



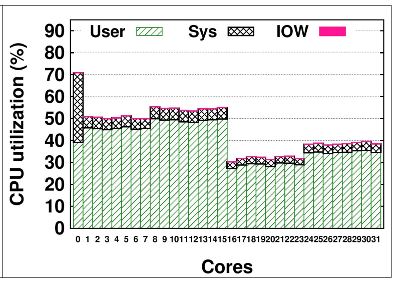
(a) HBase



(b) H-*Tucana*



(c) NW-H-*Tucana*



(d) Ideal-H-*Tucana*

Figure 4.4: CPU utilization per core at the server side during the execution of the run phase with 100M records.

put operations. Since NW-H-*Tucana* does not perform any I/O, the reason is again that gets are lock-less operations, whereas puts are not.

Ideal-H-*Tucana* outperforms H-*Tucana* by up to 2$\times$ and 14.2% during the load and run phase by avoiding lookups over the tree, insertions, and I/O operations.

NW-H-*Tucana* outperforms Ideal-H-*Tucana* by up to 23% during the run phase. The reason is that Ideal-H-*Tucana* has to do an allocation for each get operation, but the NW-H-*Tucana* version just returns an exiting value. The allocation cost more than the work done by the NW-H-*Tucana* version. Due to this allocation, with Ideal-H-*Tucana*, the load phase outperforms by up to 27% the run phase.

4.1.4.2 CPU utilization analysis

For the small dataset, Figures 4.3 and 4.4 depict, for the server, the CPU utilization per core produced during the execution of the load and run phases, respectively, with the four engines. For the client, Figure 4.5 provides the average CPU utilization for both phases, also for the smaller dataset. Figure 4.6 depicts, for HBase and H-*Tucana*, the average CPU utilization for both phases with the 500M dataset. For the server, Table 4.2 provides the average number of cores that has a CPU utilization larger than 80%, the average CPU utilization achieved, and also the percentage of time during which this saturation occurs.

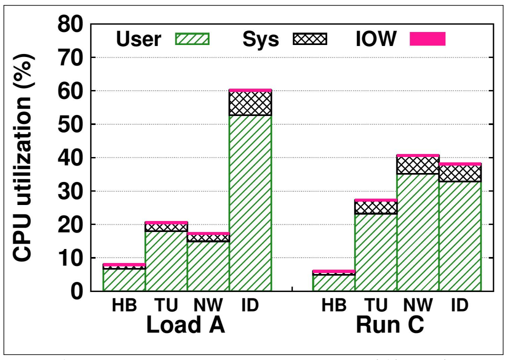


Figure 4.5: For the 100M dataset, average CPU utilization at the client side during the execution of both phases for the four engines.

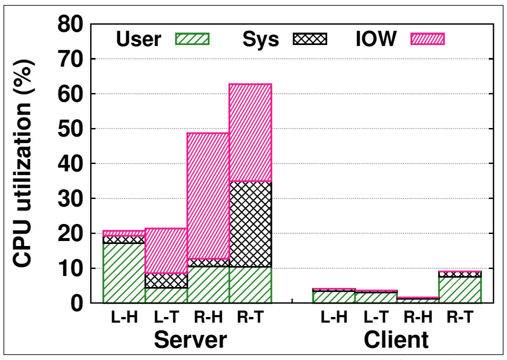


Figure 4.6: For the 500M dataset, average CPU utilization at the server and client, during the execution of the load and run phase for HBase and H-*Tucana*. L-H and R-H represent the load and run phase with HBase, and L-T and R-T represent the load and run phase with H-*Tucana*.

Regarding the 100M records dataset, for HBase, the CPU utilization is, on average, 34.6% and 55.3% for the load and run phases, respectively. However, Table 4.2 shows that, during both phases, there are several cores (two and thirteen) that achieve a CPU utilization close to or above 90%. This effect does not appear on Figures 4.3 and 4.4, because the cores saturated are not always the same. These cores become a bottleneck, and do not allow HBase to provide better performance. Some of these cores are usually executing tasks related to the network path, as the analysis for Ideal-H-*Tucana* shows. On the contrary, the client has an average CPU utilization lower than 10%.

H-*Tucana* has an average CPU utilization of 33.6% and 39.1% for the load and run phases, respectively. Being more lightweight in CPU utilization than HBase, H-*Tucana* provides a significant better performance. During the run phase, a small percentage of the CPU utilization is for I/O wait time due to the commit that occurs at the beginning of this phase. However, there is no I/O due to get operations because the dataset fits in memory. For both phases, one core presents an average CPU utilization above 80% during more of the half of the execution time. Consequently, during this time, H-*Tucana* cannot provide a better performance. As results for Ideal-H-*Tucana* show, these cores are executing tasks related to the network path (inherited from HBase). The client still provides a low CPU utilization.

NW-H-*Tucana* provides an average CPU utilization of 33.5% and 44.4% for the load and run phases, respectively. The load phase has one core with a CPU utilization of 99% during two thirds of the time. The run phase also has saturation problem, since during half of its execution time one core has a CPU utilization of 80%. It can be considered that one core becomes a bottleneck, and NW-H-*Tucana* cannot provide better performance. In addition, although the client presents a low CPU utilization, during the run phase, two cores present a CPU utilization above 80% during 70% of the execution time (this data is not presented due to space constraint).

Ideal-H-*Tucana* provides a low CPU utilization: on average, 26.1% and 23.1% for the load and run phases, respectively. But, Table 4.2 shows a saturation problem during the load phase, because the CPU utilization of one core is over 90%. Since, Ideal-H-*Tucana* does not execute any code from *Tucana*, this bottleneck appears due to HBase that mainly executes network tasks. Therefore, it can be claimed that HBase has a problem on its network path that implies a serialization of the code. The client average CPU utilization is larger than the server. Indeed, during the load phase, it is 60.1%, and four cores achieve an average CPU utilization close to 88%. Therefore, it can be considered that the client is close to saturation as well. Ideal-H-*Tucana* cannot provide a better performance when inserting KV pairs.

During the run phase, Ideal-H-*Tucana* does not have saturation problem at the server. The client presents an average CPU utilization of 40.6%, and six cores has an utilization above 80% but only 25% of the time. Therefore, during this phase, the CPU is not a bottleneck.

With the 500M dataset, Figure 4.6 shows that H-*Tucana* presents a low CPU utilization at the server during the load phase. The problem is the memory pressure because the dataset does not fit in memory, and the system spends most of the time managing pages faults. *Tucana* uses `mmap` as a single allocator for memory and device space, and *Tucana* cannot decide which pages are evicted from memory and which are kept. Therefore, pages containing index and leaf nodes are evicted, and they have to be re-read from the devices. This behavior is analyzed in subsections 4.1.4.3 and 4.1.4.4.

size	Phase	Engine	CPU util	# Cores	% Time
100M	Load	HB	89.4	2.0	94.1
		TU	88.3	1.0	61.5
		NW	99.0	1.1	68.3
		ID	90.0	1.0	65.0
	Run	HB	86.1	13.0	97.2
		TU	80.7	1.3	55.2
		NW	80.2	1.0	59.1
		ID	0.0	0.0	0.0
500M	Load	HB	90.8	1.8	96.0
		TU	86.5	1.9	66.7
	Run	HB	86.9	12.3	99.9
		TU	84.2	14.4	97.5

Table 4.2 Number of cores with a CPU utilization larger than 80%, average CPU utilization, and duration (in time percentage) of this saturation.

During the run phase, the percentage of time used for I/O wait is significantly increased due to the amount of data that has to be read for the get operations since the dataset does not fit in memory.

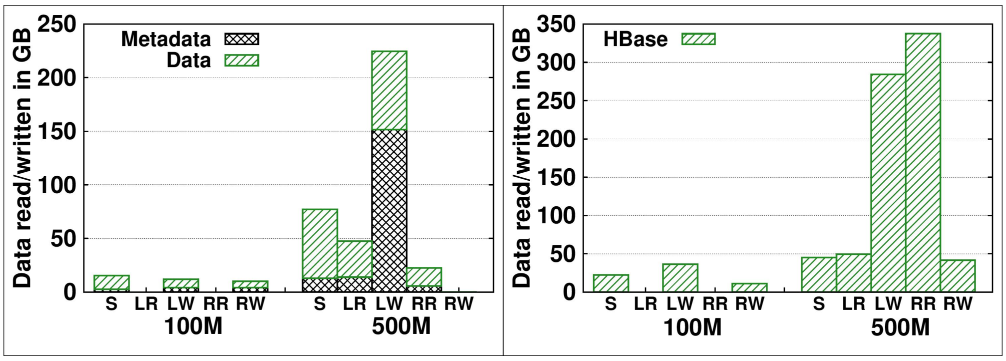
With HBase, the server presents a low average CPU utilization during the load phase, however almost two cores present more than 90% during the execution of the test. The run phase presents a higher average CPU utilization, with a high I/O wait percentage due to the I/O read operations. But, again, the CPU becomes a bottleneck with twelve cores having a CPU utilization close to 87%. Therefore, it can be claimed that HBase cannot provide better performance.

With the large dataset, the client provides a quite low CPU utilization with both HBase and H-*Tucana*. The reason is the drop of performance due to the lack of memory at the server.

4.1.4.3 I/O analysis

Figure 4.7 depicts the total size of the dataset and the amount of data read and written during the execution of the test with H-*Tucana* and HBase. For H-*Tucana*, metadata represents blocks used for internal and leaf nodes, and data represents blocks used for the KV log. For HBase, it is not able to distinguish between data and metadata.

H-*Tucana* presents a total write amplification of up to 37% and 2.2× for the 100M and 500M records, respectively. However, several things should be highlighted. During the load phase, the amount of data blocks written is almost equal to the total size of key value log. The extra data blocks written are because a block could be written several times on the device due to commits. But once a data block is full and written on the device, it is never re-written again. However, the amount of metadata blocks, written is up to 11.8× larger than the actual size of the metadata. The reason is that internal and leaf nodes are modified several times and in different instant of time. In addition, with `mmap` modified disk blocks are written to the device not only



(a) H-*Tucana* (b) HBase

Figure 4.7: Total size, in GB, of the dataset (metadata plus data) and amount of data, in GB, read and written during the execution of the test with H-*Tucana* and HBase, for both phases. S stands for Size, LR and LW for reads and writes performed during the load phase, and RR and RW for reads and writes performed during the run phase.

during *Tucana*'s commit operations, but also periodically, by the flush kernel threads when they are older than a threshold or when free memory shrinks below a threshold, using an LRU policy and `madvise` hints. This policy also increases the times a metadata block is written on the device.

During the run phase, the blocks written to the device are due to the *Tucana*'s commit operations issued at the beginning of this phase. This commit writes on the device blocks modified at the end of the load phase but not written to the device yet.

During the load phase, with 100M records, no data is read from the storage device. However, with 500M records, metadata blocks are read because internal or leaf nodes are evicted from memory. On the contrary, the data blocks are read to make key comparisons. The problem is that `mmap` controls which pages are evicted from memory due to memory pressure, and the current implementation of *Tucana* cannot modify this behavior. As a result, `mmap` evicts not only data pages, but also metadata pages. This behavior reduces the amount of I/Os that can be amortized for inserts due to the limited buffering in the B^{ε}–tree.

During the run phase, no data is read with the small dataset. However, the large dataset implies larger amount of data read from the SSDs, since the whole dataset does not fit in memory.

HBase presents a write amplification problem due to the compactions performed that is shown in Figure 4.7. Its total write amplification is up to $2.1\times$ and $7.1\times$ for the 100M and 500M records, respectively. With respect to the amount of data read, HBase does not read any data with the smaller dataset. With the large dataset, HBase reads up to $8.5\times$ the size of the dataset.

Tucana has better inherent behavior with respect to amplification thanks to its B^{ε}–tree. Indeed, *Tucana* does not require compactions at the expense of more random and small I/Os. HBase introduces compactions, and ends up issuing larger requests. With 500M records, during the load phase, the average request size is 239 kB and 641 kB for H-*Tucana* and HBase, respectively. Consequently, although HBase writes by up to 45% more data than H-*Tucana*, HBase outperforms H-*Tucana* thanks to its

I/O access pattern. With 100M records, during the load phase, the average request size is 493 kB and 478 kB for H-*Tucana* and HBase, respectively, and therefore, there is almost no difference in I/O performance between them.

4.1.4.4 Network and I/O traffic analysis

For H-*Tucana*, Figure 4.8 depicts the throughput and network traffic during the execution of both phases, whereas Figure 4.9 depicts the throughput and I/O traffic.

Figure 4.8 shows that the network devices are not a bottleneck, since they are able to provide up to 40 Gbit/s, but the maximum in-coming throughput achieved is 380 MB/s during the load phase, and the maximum out-going throughput achieved is 853 MB/s during the run phase. Therefore, as subsection 4.1.4.2 says, the network path of HBase does not allow *Tucana* to achieve better network throughput.

Regarding I/O traffic, with the small dataset, the storage devices provides up to 515 MB/s of write throughput, achieving a 100% of disk utilization. Therefore, in this case, *Tucana* is achieving almost its maximum performance, with high utilization, and the devices can become a bottleneck. During the run phase, there is write traffic due to the commit. Both phases do not have read traffic.

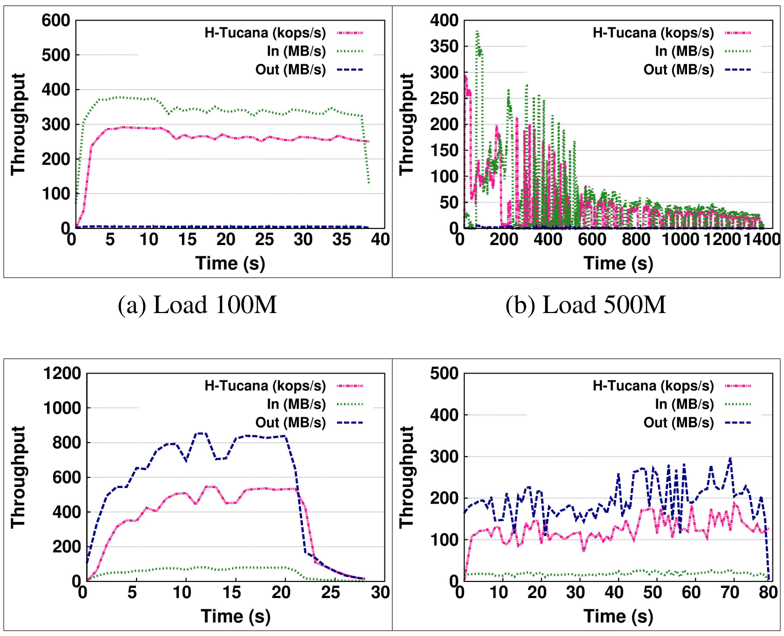
With the 500M records dataset, during the load phase storage devices provides up to 512 MB/s of write throughput, however, the write performance drops as the system runs out of memory, and achieves on average 163 MB/s. Regarding reads, at the beginning of the execution there is not read traffic, since the dataset still fits in memory. However, when the dataset does not fit in memory, the read traffic is significantly increased. Due to the read and write traffic, the storage device is 100% utilized. During the run phase, the storage device provides up to 378 MB/s of read throughput, and it is close to a 100% of disk utilization.

4.1.5 Summary

This section analyzes host CPU overhead, network, storage and memory limitations of an efficient key-value store, *Tucana*, designed for fast storage devices. *Tucana* is a feature-rich key-value store that supports variable size keys and values, versions, arbitrary data set sizes, concurrency, multithreading, and persistence and recovery from failures. The issues analyzed include host CPU overhead, network and I/O traffic, and memory use.

4.2 Data-centric workflow runtime for data-intensive applications on Cloud systems

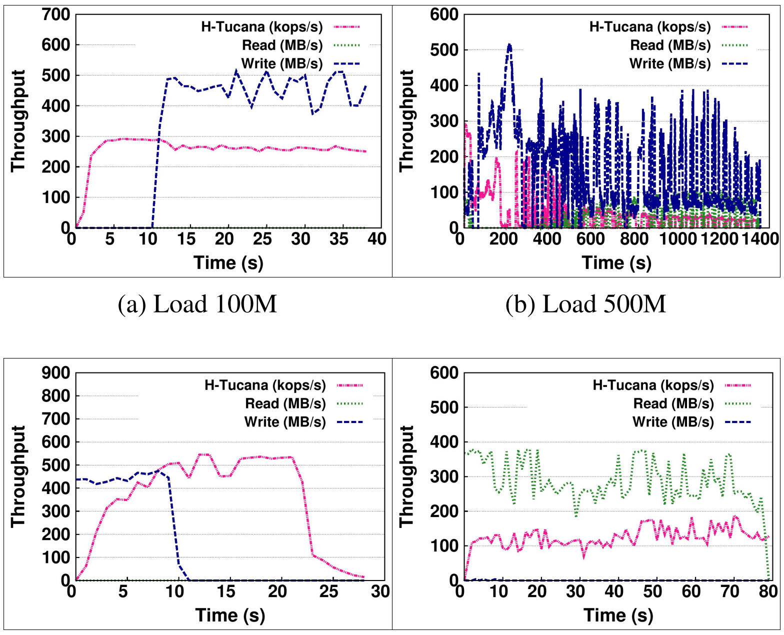
In the last decade, the scientific computing scenario is greatly evolving in two main areas. First, the focus on scientific computation is changing from CPU-intensive jobs, like large scale simulations or complex mathematical applications, towards a data-intensive approach. This new paradigm greatly affects the underlying architecture requirements, slowly vanishing the classical CPU bottleneck and exposing bottlenecks in the I/O systems. Second, the evolution in computing technologies and science



(a) Load 100M (b) Load 500M

(c) Run 100M (d) Run 500M

Figure 4.8: For H-*Tucana*, throughput (kop/s) and network traffic (MB/s) during the execution of the load and run phase for 100M and 500M records. In and Out stand for input and output network traffic, respectively.



(a) Load 100M (b) Load 500M

(c) Run 100M (d) Run 500M

Figure 4.9: For H-*Tucana*, throughput (kop/s) and I/O traffic (MB/s) during the execution of the load and run phase for 100M and 500M records.

funding restrictions are changing the available computing resources in the scientific community.

In current approaches, the interfaces and management solutions of the different infrastructures present notable differences, requiring different programming models, even for the same application. On the other side, the future ultrascale systems should take advantage of every possible resource available in a transparent way for the user [171].

Workflow management systems are computing platforms widely used today for designing and executing data-intensive applications over High-Performance Computing (HPC) systems or distributed infrastructures. Data-intensive workflows consist of interdependent data processing tasks, often connected in a DAG style, which communicate through intermediate storage abstractions, typically files [172].

Current trends in scientific computing and data-intensive applications are involved in the use of Cloud infrastructures as a flexible approach to virtually limitless computing resources in a pay-per-use basis. Additionally, several research centers complement their private computing infrastructure (usually HPC systems) with public Cloud resources. The systems from both HPC and Cloud domains are not efficiently supporting data-intensive workflows, especially because their design was thought for individual applications and not for ensembles of cooperating applications. Given this current scenario, a solution that combines characteristics typical of HPC, data analysis, and Cloud computing is becoming more and more necessary.

This section describes a data-aware scheduling strategy [173] for exploiting data locality in data-intensive workflows executed over the DMCF [174] and Hercules [175] platforms. Some experimental results are presented to show the benefits of the proposed data-aware scheduling strategy for executing data analysis workflows and for demonstrating the effectiveness of the solution. Using a data-aware strategy and Hercules as temporary storage service, the I/O overhead of workflow execution has been reduced by 55% compared to the standard execution based on the Azure storage Cloud infrastructure, leading to a 20% reduction of the total execution time. This evaluation confirms that our data-aware scheduling approach is effective in improving the performance of data-intensive workflow execution in Cloud platforms.

Some existing solutions focused on the use of in-memory storage as a different approach for solving the bottlenecks in highly concurrent I/O operations, such as Parrot and Chirp [176]. Our in-memory approach takes hints from all these solutions, by i) offering popular data access APIs (POSIX, put/get, MPI-IO), ii) focusing on flexible scalability through distributed approaches for both data and metadata, iii) the use of compute nodes (or worker VMs) to enhance the I/O infrastructure, and iv) facilitating the deployment of user-level components.

The remainder of this section is structured as follows. Subsection 4.2.1 describes the main features of DMCF. Subsection 4.2.2 introduces Hercules architecture and capabilities. Subsection 4.2.3 emphasizes the advantages of integrating DMCF and Hercules. Subsection 4.2.4 details the data-aware scheduling technique proposed. Subsection 4.2.5 presents the experimental results. Finally, Subsection 4.2.6 concludes this section.

4.2.1 Data Mining Cloud Framework overview

The Data Mining Cloud Framework (DMCF) [174] is a software system implemented for designing and executing data analysis workflows on Clouds. A Web-based user interface allows users to compose their applications and submit them for execution over Cloud resources, according to a Software-as-a-Service (SaaS) approach.

The DMCF architecture has been designed to be deployed on different Cloud settings. Currently, there are two different deployments of DMCF: *i*) on top of a Platform-as-a-Service (PaaS) Cloud, i.e., using storage, compute, and network APIs that hide the underlying infrastructure layer; *ii*) on top of an Infrastructure-as-a-Service (IaaS) Cloud, i.e., using virtual machine images (VMs) that are deployed on the infrastructure layer. In both deployment scenarios, we use Microsoft Azure[13] as Cloud provider.

The DMCF software modules can be grouped into *web components* and *compute components* (see top-left part of Figure 4.10). DMCF allows users to compose, check, and run data analysis workflows through a HTML5 web editor. The workflows can be defined using two languages: *VL4Cloud* (Visual Language for Cloud) [174] and *JS4Cloud* (JavaScript for Cloud) [177]. Both languages use three key abstractions:

- *Data* elements, representing input files (e.g., a dataset to be analyzed) or output files (e.g., a data mining model).
- *Tool* elements, representing software tools used to perform operations on data elements (partitioning, filtering, mining, etc.).
- *Tasks*, which represent the execution of Tool elements on given input Data elements to produce some output Data elements.

The DMCF editor generates a JSON descriptor of the workflow, specifying what are the tasks to be executed and the dependency relationships among them. The JSON workflow descriptor is managed by the DMCF workflow engine that is in charge of executing workflow tasks on a set of workers (virtual processing nodes) provided by the Cloud infrastructure. The workflow engine implements a data-drive task parallelism that assigns workflow tasks to idle workers as soon as they are ready to execute. Further details on DMCF execution mechanisms are given in Subsection 4.2.3.

4.2.2 Hercules overview

Hercules [175] is a distributed in-memory storage system based on the key/value Memcached database. The distributed memory space can be used by the applications as a virtual storage device for I/O operations. Hercules has been adapted for being used as an alternative to Cloud storage service, offering in-memory shared storage for applications deployed over Cloud infrastructures.

Hercules architecture (see top-center part of Figure 4.10, labeled Hercules instance) has two main layers: front-end (Hercules client library) and back-end (server layer). The user-level library is used by the application (or DMCF workers) for accessing to the Hercules back-end. The library features a layered design, while

[13]http://azure.microsoft.com

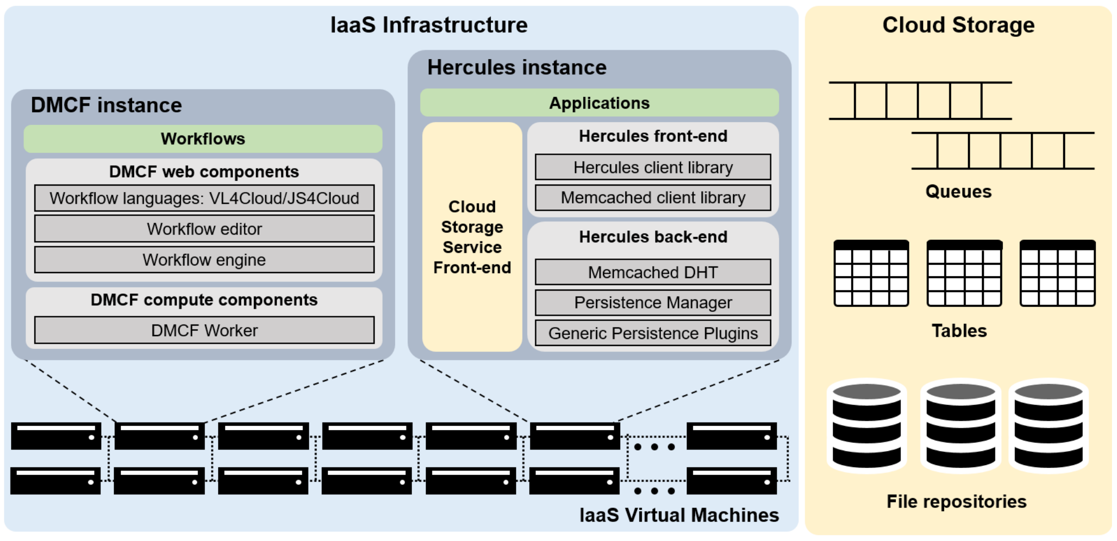


Figure 4.10: DMCF+Hercules architecture. Under an Iaas infrastructure, DMCF acts as a workflow engine while Hercules accelerates I/O accesses.

back-end components are based on enhanced Memcached servers that extend basic functionality with persistence and tweaks.

Hercules offers four main advantages: scalability, easy deployment, flexibility, and performance.

Scalability is achieved by fully distributing data and metadata information among all the available nodes, avoiding the bottlenecks produced by centralized metadata servers. Data and metadata placement is completely calculated at client-side by a hashing algorithm. The servers are completely stateless.

Easy deployment and flexibility at worker-side is provided by a POSIX-like user-level interface in addition to the classic put/get approach existing in current NoSQL databases. This approach supports legacy applications with minimum changes. Servers can also be deployed without root privileges.

Performance and flexibility are targeted at server-side by exploiting I/O parallelism. The capacity of dynamically deploying as many Hercules nodes as necessary provides the flexibility feature. The combination of both approaches results on each node being accessed independently, multiplying the total throughput peak performance.

4.2.3 Integration between DMCF and Hercules

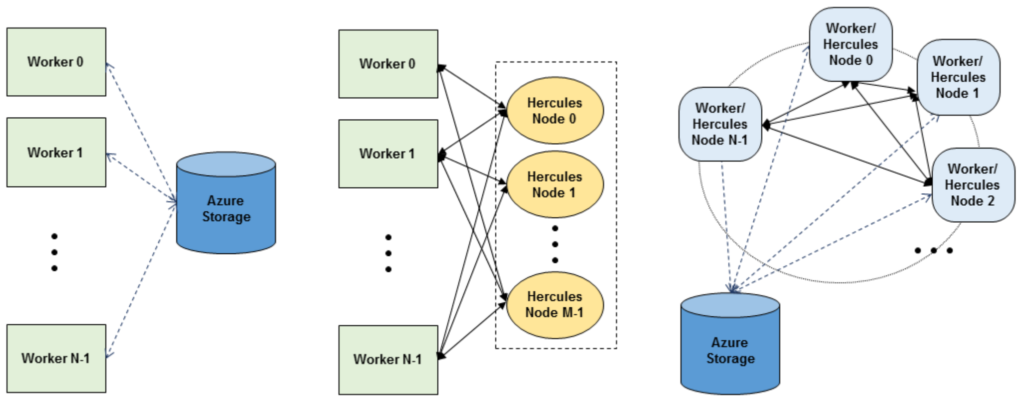
DMCF and Hercules can be configured to achieve different levels of integration, as shown in Figure 4.11.

Figure 4.11a shows the original approach of DMCF, where every I/O operation is carried out against the Cloud storage service offered by the Cloud provider (Azure Storage). There are, at least, four disadvantages about this approach: usage of proprietary interfaces, I/O contention in the service, lack of configuration options, and persistence-related costs unnecessary for temporary data. Figure 4.11b presents a second scenario based on the use of Hercules as the default storage for temporary generated data. Hercules I/O nodes can be deployed on as many VM instances as needed by the user depending on the required performance and the characteristics of data. Figure 4.11c shows a third scenario with a tighter integration of DMCF and Hercules infrastructures [178]. In this scenario, initial input and final output are stored on persistent Azure storage, while intermediate data are stored on Hercules in-memory nodes. Hercules I/O nodes share virtual instances with the DMCF workers.

Current trends in data-intensive applications are extensively focusing on the optimization of I/O operations by exploiting the performance offered by in-memory computation, as shown by the popularity of Apache Spark Resilient Distributed Datasets (RDD). The goal of this subsection is strengthening the integration between DMCF and Hercules by leveraging the co-location of compute workers and I/O nodes for exposing and exploiting data locality. In order to simplify the implementation of the solution, some workarounds were used: each time that one worker needed to access data (read/write operations over a file), it copied the whole file from Hercules servers to the worker local storage. This approach may greatly penalize the potential performance gain in I/O operations for two main reasons:

- *Data placement strategy*. The original Hercules data placement policy distributes every partition of a specific file among all the available servers. This strategy has two main benefits: avoids hot spots and improves parallel accesses. In an improved DMCF-Hercules integration, whole files can be stored on the same Hercules server.
- *Data locality agnosticism*. Data-locality will not be fully exploited until the DMCF scheduler is tweaked for running tasks on the node that contains the necessary data and/or the data is placed where the computation will be realized.

Figure 4.12 describes an improvement to the third scenario of integration between DMCF and Hercules, which is exploited as the base for the proposed data-aware scheduling strategy. Four main components are present: DMCF worker daemon, Hercules daemon, Hercules client library, and Azure client library. The DMCF worker daemons are in charge of executing the workflow tasks; Hercules daemons



(a) Scenario 1: Azure storage. (b) Scenario 2: Hercules. (c) Scenario 3: Azure storage and Hercules.

Figure 4.11: Integration scenarios between DMCF and Hercules.

act as I/O nodes (storing data in-memory and managing data accesses); the Hercules client library is intended to be used by the applications to access to the data stored in Hercules (query Hercules daemons); the Azure client library is used to read/write data from/to the Azure storage.

In this model, we use a RAM disk as generic storage buffer for I/O operations performed by workflow tasks. The objective of this approach is to enable the support of DMCF to any existing tool, allowing even binaries independently of the language used for their implementation, while offering in-memory performance for local accesses.

The logic used for managing this RAM disk buffer is based on the full information about the workflow possessed by the DMCF workers. When every dependency of an specific task is fulfilled (every input file is ready to be accessed) the DMCF worker brings the necessary data to the node from the storage (Azure Storage in the first scenario or Hercules in the second scenario).

Based on this approach, every existing tool is capable of accessing transparently data, without the need of modifying the code. After the termination of the task, every data written in the RAM disk is transferred to Hercules/Azure by the DMCF worker. Every data transfer to/from Hercules is performed by the DMCF worker through the Hercules client library.

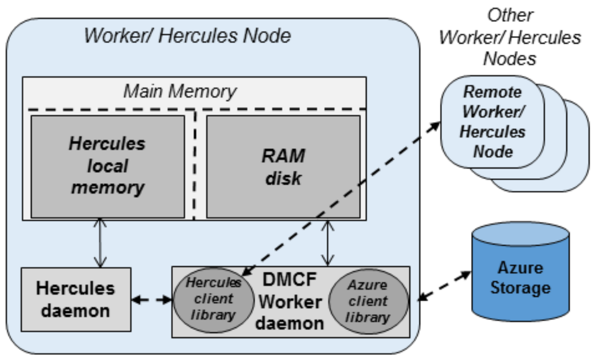


Figure 4.12: DMCF and Hercules daemons.

4.2.4 *Data-aware scheduling strategy*

This subsection presents the data-aware scheduling strategy that combines DMCF load-balancing capabilities and Hercules data and metadata distribution functionalities for implementing various locality-aware and load-balancing policies.

Before going into details of the purposed scheduling mechanism, we recall the high-level steps that are performed for designing and executing a knowledge discovery application in DMCF [174]:

1. The user accesses the website and designs the workflow through a Web-based interface.
2. After workflow submission, the system creates a set of tasks and inserts them into the Task Queue.

3. Each idle worker picks a task from the Task Queue, and concurrently executes it.
4. Each worker gets the input dataset from its location. To this end, a file transfer is performed from the Data Folder where the dataset is located, to the local storage of the worker.
5. After task completion, each worker puts the result on the Data Folder.
6. The Website notifies the user whenever each task has completed, and allows her/him to access the results.

```
1  Procedure main()
2      while true do
3          if locallyActivatedTasks.isNotEmpty() or TaskQueue.isNotEmpty() then
4              task ← selectTask(locallyActivatedTasks, TaskQueue);
5              TaskTable.update(task, ‘running’);
6              foreach input in task.inputList do
7                  transfer(input, DataFolder, localDataFolder);
8              transfer(task.tool.executable, localToolFolder);
9              foreach library in task.tool.libraryList do
10                 transfer(library, ToolFolder, localToolFolder);
11             taskStatus ← execute(task, localDataFolder, localToolFolder);
12             if taskStatus = ‘done’ then
13                 foreach output in task.outputList do
14                     transfer(output, localDataFolder, DataFolder);
15                 TaskTable.update(task, ‘done’);
16                 foreach wfTask in TaskTable.getTasks(task.workflowId) do
17                     if wfTask.dependencyList.contains(task) then
18                         wfTask.dependencyList.remove(task);
19                         if wfTask.dependencyList.isEmpty() then
20                             TaskTable.update(wfTask, ‘ready’);
21                             locallyActivatedTasks.addTask(wfTask);
22             else //Manage the task’s failure;
23         else //Wait for new tasks in TaskQueue;
24 SubProcedure selectTask(locallyActivatedTasks, TaskQueue)
25     bestTask ← Hercules.selectByLocality(locallyActivatedTasks, TaskQueue);
26     if bestTask in TaskQueue then
27         TaskQueue.removeTask(bestTask);
28     foreach task in locallyActivatedTasks do
29         if task != bestTask then
30             TaskQueue.addTask(task);
31     empty(locallyActivatedTasks);
32     clean(localDataFolder);
33     clean(localToolFolder);
34     return bestTask;
```

Algorithm 1. Worker operations.

Algorithm 1 describes the data-aware scheduler employed by each worker. Compared to the original scheduler described in [174], this novel scheduling algorithm relies on a local list within each worker, called *locallyActivatedTasks*. This list contains all the tasks whose status was changed by this worker from ‘*new*’ to ‘*ready*’ at the end of the previous iteration. The worker cyclically checks whether there are tasks ready to be executed in *locallyActivatedTasks* or in the Task Queue (lines

2-3). If so, a task is selected from one of the two sets (line 4) using the *selectTask* subprocedure (lines 24-34), and its status is changed to '*running*' (line 5). Then, the transfer of all the needed input resources (files, executables, and libraries) is carried out from their original location to two local folders, referred to as *localDataFolder* and *localToolFolder* (lines 6-10). At line 11, the worker locally executes the *task* and waits for its completion. If the *task* is '*done*' (line 12), if necessary the output results are copied to a remote data folder (lines 13-14), and the *task* status is changed to '*done*' also in the Task Table (line 15). Then, for each *wfTask* that belongs to the same workflow of *task* (line 16), if *wfTask* has a dependency with *task* (line 17), that dependency is deleted (line 18). If *wfTask* remains without dependencies (line 19), it becomes '*ready*' and is added to the Task Queue (lines 20-21). If the *task* fails (line 22), all the tasks that directly or indirectly depend on it are marked as '$failed$'.

The *selectTask* subprocedure works as follows. First, it invokes the *selectByLocality* function provided by Hercules, which selects the best task that this worker can manage from *locallyActivatedTasks* and the Task Queue (line 25). To take advantage of data locality, the best task selected by this function is the one having the highest number of input data that are available on the local storage of the worker, based on the information available to Hercules. This differs from the original data-locality agnostic scheduling policy adopted in DMCF, in which each worker picks and executes the task from the queue following a FIFO policy. If such best task was chosen from the Task Queue, then that task is removed from the Task Queue (line 26-27). All the tasks in *locallyActivatedTasks* that are different from the best task are added to the Task Queue, thus allowing the other workers to execute them (line 28-30). Finally, *locallyActivatedTasks* is emptied, and localDataFolder and localToolFolder are cleaned from the data/tools that are not used by bestTask (lines 31-33).

4.2.5 Experimental evaluation

This subsection presents the experimental evaluation of the data-aware scheduling strategy used to improve the integration between DMCF and Hercules. For this evaluation, we have emulated the execution of a data analysis workflow using three alternative scenarios:

- *Azure-only*: every I/O operation of the workflow is performed by DMCF using the Azure storage service.
- *Locality-agnostic*: a full integration between DMCF and Hercules is exploited, where each intermediate data is stored in Hercules, while initial input and final output are stored on Azure. DMCF workers and Hercules I/O nodes share resources (they are deployed in the same VM instance), however, every I/O operation is performed over remote Hercules I/O nodes through the network.
- *Data-aware*: based on the same deployment as in the previous case, this scenario is based on a full knowledge of the data location, and executes every task in the same node where the data are stored, leading to fully local accesses over temporary data. Based on this locality exploitation, most I/O operations are performed in-memory rather than through the network.

Listing 4.1: Classification JS4Cloud workflow.

```
 1: var TrRef = Data.get("Train");
 2: var STrRef = Data.define("STrain");
 3: Shuffler({dataset:TrRef, sDataset:STrRef});
 4: var n = 20;
 5: var PRef = Data.define("TrainPart", n);
 6: Partitioner({dataset:STrRef, datasetPart:PRef});
 7: var MRef = Data.define("Model", [3,n]);
 8: for(var i = 0; i <n; i++){
 9:    C45({dataset:PRef[i], model:MRef[0][i]});
10:    SVM({dataset:PRef[i], model:MRef[1][i]});
11:    NaiveBayes({dataset:PRef[i], model:MRef[2][i]});
12: }
13: var TeRef = Data.get("Test");
14: var BMRef = Data.define("BestModel");
15: ModelSelector({dataset:TeRef, models:MRef, bestModel:BMRef});
16: var m = 80;
17: var DRef = Data.get("Unlab", m);
18: var FDRef = Data.define("FUnlab", m);
19: for(var i = 0; i <m; i++)
20:    Filter({dataset:DRef[i], fDataset:FDRef[i]});
21: var CRef = Data.define("ClassDataset", m);
22: for(var i = 0; i <m; i++)
23:    Predictor({dataset:FDRef[i], model:BMRef, classDataset:CRef[i]});
```

The evaluation is based on a data mining workflow that analyzes *n* partitions of the training set using *k* classification algorithms so as to generate *kn* classification models. The *kn* models generated are then evaluated against a test set by a model selector to identify the best model. Then, *n* predictors use the best model to produce in parallel *n* classified datasets. The *k* classification algorithms used in the workflow are C4.5, Support Vector Machine (SVM) and Naive Bayes, which are three of the main classification algorithms [179]. The training set, test set and unlabeled dataset that represent the input of the workflow, have been generated from the *KDD Cup 1999*'s dataset[14], which contains a wide variety of simulated intrusion records in a military network environment.

Listing 4.1 shows the JS4Cloud source code of the workflow. At the beginning, we define the training set (*line 1*) and a variable that stores the shuffled training set (*line 2*). At *line 3*, the training set is processed by a shuffling tool. Once defined parameter $n = 20$ at *line 4*, the shuffled training set is partitioned into *n* parts using a partitioning tool (*line 6*). Then, each part of the shuffled training set is analyzed in parallel by $k = 3$ classification tools (*C*4.5, *SVM*, *NaiveBayes*). Since the number of tools is *k* and the number of parts is *n*, *kn* instances of classification tools run in parallel to produce *kn* classification models (*lines 8-12*). The *kn* classification models generated are then evaluated against a test set by a model selector to identify the best model (*line 15*). Then, $m = 80$ unlabeled datasets are specified as input (*line 15*). Each of the *m* input datasets is filtered in parallel by *m* filtering tools (*lines 19-20*). Finally, each of the *m* filtered datasets is classified by *m* predictors using the best

[14]http://kdd.ics.uci.edu/databases/kddcup99/kddcup99

model (*lines 22-23*). The workflow is composed of $3+kn+2m$ tasks. In the specific example, where $n=20$, $k=3$, $m=80$, the number of generated tasks is equal to 223.

Figure 4.13 depicts the VL4Cloud version of the data mining workflow. The visual formalism clearly highlights the level of parallelism of the workflow, expressed by the number of parallel paths and the cardinality of tool array nodes.

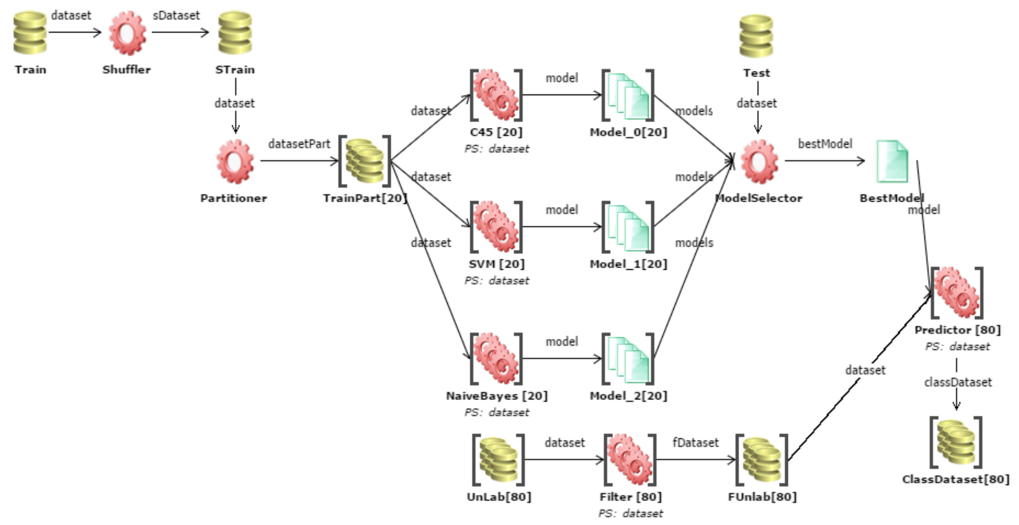


Figure 4.13: Classification VL4Cloud workflow.

Once the workflow is submitted to DMCF using either JS4Cloud or VL4Cloud, DMCF generates a JSON descriptor of the workflow, specifying which are the tasks to be executed and the dependency relationships among them. Thus, DMCF creates a set of tasks that will be executed by workers. In order to execute a given workflow task, we have provisioned as many D2 VM instances (CPU with 2 cores and 7GB of RAM) in the Azure infrastructure as needed, and configured them by launching both the Hercules daemon and the DMCF worker process on each VM. In order to better understand the performance results, we have performed a preliminary evaluation of the expected performance of both Azure Storage and Hercules. Table 4.3 presents bandwidth performance of an I/O micro-benchmark consisting in writing and reading a 256 MB file, with 4 MB chunk size. Latency results are not relevant for data-intensive applications. Due to the usual large file size, the latency-related time is negligible in comparison with the bandwidth-related time.

Table 4.3 Bandwidth micro-benchmark results (in MB/s).

Operation	Hercules local access	Hercules remote access	Azure Storage data access
Read	800	175	60
Write	1,000	180	30

After the initial setup, DMCF performs a series of preliminary operations (i.e., getting the task from the Task Queue, downloading libraries, and reading input files from the Cloud storage) and final operations (e.g., updating the Task Table, writing the output files to the Cloud storage). Table 4.4 lists all the read/write operations performed during the execution of the workflow on each data array. Each row of the table describes: *i*) the number of files included in the data array node; *ii*) the total size of the data array; *iii*) the total number of read operations performed on the files included in the data array; and *iv*) the total number of write operations performed on the files included in the data array. As can be noted, all the inputs of the workflow (i.e., *Train*, *Test*, *UnLab*) are never written on persistent storage, and the final output of the workflow (i.e., *ClassDataset*) is not used (read) by any other task.

Table 4.4 Read/write operations performed during the execution of the workflow.

Data node	N. of files	Total size	N. of read operations	N. of write operations
Train	1	100MB	1	-
Strain	1	100MB	1	1
TrainPart	20	100MB	60	20
Model	60	$\approx$20MB	60	60
Test	1	50MB	1	-
BestModel	1	300KB	80	1
UnLab	80	8GB	80	-
FUnLab	80	$\approx$8GB	80	80
ClassDataset	80	$\approx$6GB	-	80

Figure 4.14a shows the turnaround times of the workflow executed in the three scenarios introduced earlier: *Azure-only*, *Locality-agnostic*, *Data-aware*. In all scenarios, the turnaround times have been measured varying the number of workers used to run it on the Cloud from 1 (sequential execution) to 64 (maximum parallelism). In the Azure scenario, the turnaround time decreases from 1 hour and 48 minutes on a single worker, to about 2.6 minutes using 64 workers. In the Locality-agnostic scenario, the turnaround time decreases from 1 hour and 34 minutes on a single worker, to about 2.2 minutes using 64 workers. In the Data-aware scenario, the turnaround time decreases from 1 hour and 26 minutes on a single worker, to about 2 minutes using 64 workers. It is worth noticing that, in all the configurations evaluated, locality-agnostic allowed us to reduce the execution time in about 13% compared to the Azure scenario, while data-aware allowed us to reduce the execution time in about 20% compared to the Azure scenario.

We also evaluated the overhead introduced by DMCF in the three scenarios. We define as overhead the time required by the system to perform a series of preliminary operations (i.e., getting the task from the Task Queue, downloading libraries and reading input files from the Cloud storage) and final operations (e.g., updating the Task Table, writing the output files to the Cloud storage) related to the execution of each workflow task. Table 4.5 shows the overhead time of the workflow in the three analyzed scenarios. We observe that the overhead in the Azure-only scenario is 40 minutes, while in the Locality-agnostic scenario is 26 minutes and in the Data-aware is 18 minutes. This means that using Hercules for storing intermediate data we were able to reduce the overhead by 36% using a locality-agnostic approach, and by 55% using a data-aware approach.

Table 4.5 Overhead in the three scenarios.

	Total time (sec.)	Overhead (sec.)
Azure-only	6,487	2,382
Locality-agnostic	5,624	1,519
Data-aware	5,200	1,086

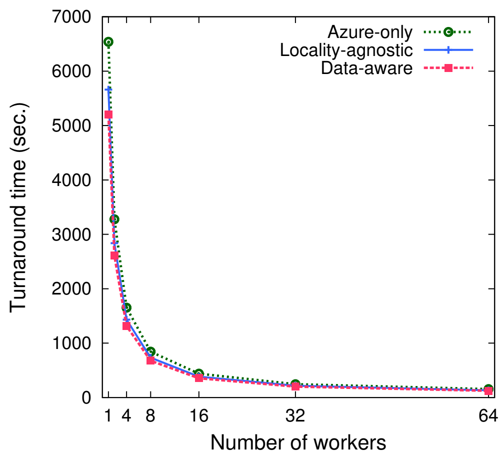
Figure 4.14b presents the time required by the application to perform every I/O operation of the application, and Figure 4.14c increases the level of detail, showing only the operations affected by the use of the Hercules service, i.e., I/O operations that work on temporary data. As shown in the figure, the difference between the three strategies is clear.

In order to better show the impact of the Hercules service, Figure 4.14d presents a breakdown of the total execution time, detailing the time spent on each of the tasks executed by the workflow application: computation tasks (CPU Time), I/O tasks over input/results files stored in Azure Storage, and I/O operations performed over temporary files. This figure clearly shows how the time required for I/O operations over temporary files, the only operations affected by use of Hercules services, are reduced to be almost negligible during the execution of the workflow, showing a great potential for increasing the I/O performance in data-intensive applications with large amounts of temporary data.

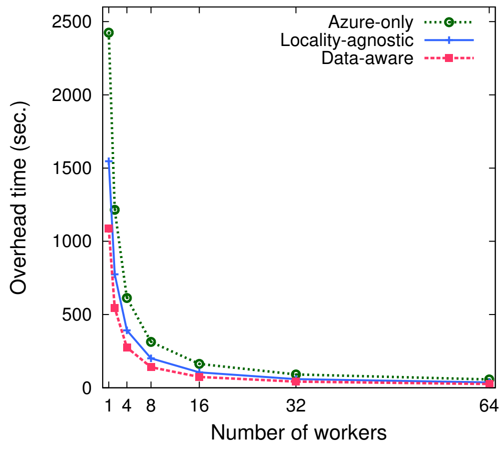
4.2.6 Conclusions

While workflow management systems deployed on HPC systems (e.g., parallel machines) typically exploit a monolithic parallel file system that ensures highly efficient data accesses, workflow systems implemented on a distributed infrastructure (most often, a public Cloud) must borrow techniques from the Big Data computing (BDC) field, such as exposing data storage locality and scheduling work to reduce data movement in order to alleviate the I/O subsystems under highly demanding data access patterns.

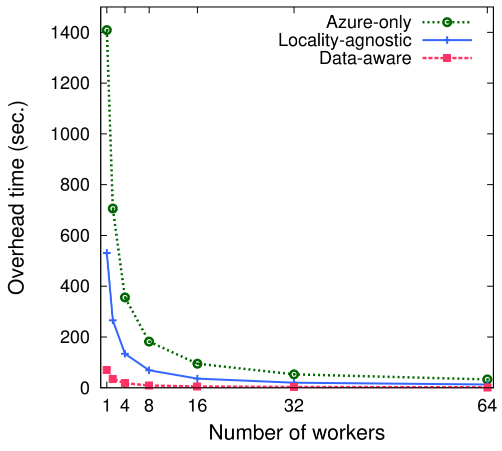
In this section, we proposed the use of DMCF+Hercules as an alternative to classical shared parallel file systems, currently deployed over the HPC infrastructures. Classical HPC systems are composed by compute resources and I/O nodes completely decoupled. Based on this approach, the available I/O nodes can scale with the number of compute nodes performing concurrent data accesses, avoiding the bottleneck of current parallel file systems. In extreme cases, our solution should also be provided over the I/O infrastructure of the parallel file system, taking advantage of the available resources (I/O nodes counting with large amounts of RAM and a dedicated high-performance network). DMCF+Hercules aims to achieve scalability to enhance four main aspects that contribute to the file I/O bottleneck illustrated as motivation above: metadata scalability, data scalability, locality exploitation, and file system server scalability. DMCF+Hercules will act as an I/O accelerator, providing improved performance for data accesses to temporary data.



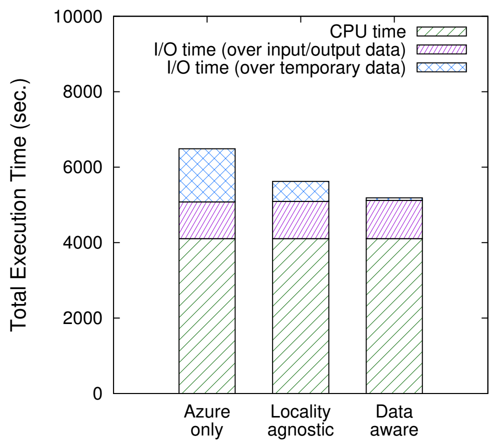
(a) Turnaround time vs number of available workers.



(b) Overhead time vs number of available workers.



(c) Overhead time vs number of available workers of only tasks that read/write temporary data.



(d) Breakdown of the total execution time.

Figure 4.14: Performance evaluation of the classification workflow using Azure-only, locality-agnostic and data-aware configurations.

This section also presented an experimental evaluation performed to analyze the performance of the proposed data-aware scheduling strategy executing data analysis workflows and to demonstrate the effectiveness of the solution. The experimental results show that, using the proposed data-aware strategy and Hercules as storage service for temporary data, the I/O overhead of workflow execution is reduced by 55% compared to the standard execution based on the Azure storage, leading to a 20% reduction of the total execution time. These results demonstrate the effectiveness of our data-aware scheduling approach in improving the performance of data-intensive workflow execution in Cloud platforms.

Acknowledgment

This section is partially supported by EU under the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS). This section has been partially supported by the Spanish MINISTERIO DE ECONOMÍA Y COMPETITIVIDAD under the project grant TIN2016-79637-P "Towards unification of HPC and Big Data paradigms".

4.3 Advanced Conflict-free Replicated DataTypes

One approach to achieve ultra-scale data management is to use a full data replication paradigm with a relaxed consistency model. This is advocated given the tradeoffs of Availability, Consistency, and network Partition addressed by the CAP theorem [180]. While a relaxed consistency model allows for prompt local updates, this can lead to potential conflicts in the data once merged elsewhere. With the premise to eventually converge to a single state, manual and case-tailored solutions are unproven correct and cumbersome to use. In this section, we present a more generic and mathematically proven method though Conflict-free Replicated DataTypes [181] that guarantee eventual convergence. We present four variants of CRDT models: operation-based, pure operation-based, state-based, and delta-state based CRDTs [181, 97, 182, 98, 183]. We aim to keep the presentation simple by addressing a common "set" datatype example throughout all CRDT variants to show their differences. We finally present a case study, on the *dataClay* [184, 185] distributed platform, demonstrating how CRDTs can be used in practice[15].

4.3.1 Scalability and Availability Tradeoffs

With the immense volumes of data generated by social networks, Internet of Things, and data science, scalability becomes one of the major data management challenges. To sustain such data volumes, a data management service must guarantee both a large data storage capacity and high availability. Although the former can be increased through scaling up, i.e., augmenting the storage capacity of a service though adding larger hard drives or using RAID technology, the availability challenge remains, due

[15]The research leading to these results has received funding from the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732505, LightKone project.

to bottlenecks and single points of failure. A typical alternative is to scale out through distributing, a.k.a., replicating, the data over distinct nodes either within a small proximity, like a cluster, or scattered over a large geographical space. This however raises a challenge on the quality of data, subject to the speed of Read/Write operations and data freshness, and often governed through a data consistency model [186].

Traditionally, the common approach was to fully replicate the data and use a strong consistency model, e.g., sequential consistency or total order protocols like quorum-based consensus [187, 188]. However, the overhead of synchronization becomes intolerable under scale, especially in a geographically distributed setting or loosely coupled systems. In fact, the CAP theorem [180, 189] forces to choose between (strict) data consistency and availability, given that network partitions are hard to avoid in practice. Consequently, the recent trend is to adopt a relaxed data consistency model that trades strict consistency for high availability. This is advocated by applications that cannot afford large response times on *reads* and *writes*, and thus allow for stale reads as long as propagating local writes eventually lead to convergence [190]—assuming system quiescence.

The weakest consistency models that guarantee convergence often adopt a variant of Eventual Consistency [190] (EC) in which updates are applied locally, without prior synchronization with other replicas, such that they are eventually delivered and applied by all replicas. In practice, many applications require stronger guarantees on time and order, and advocate the causal consistency model, which enforces a *happens-before* relation [191]: A is delivered before B if A occurred before B on the same machine; or otherwise, A occurred before a *send* event and B occurred after a *receive* event (possibly by transitivity). To understand the need for causal consistency in applications, consider an example on a replicated messaging service where Bob commented on Alice's message, i.e., Alice's message happened before Bob's comment. Since, in a distributed setting, it can happen that different users read from different servers (a.k.a., replicas), without enforcing causal consistency some users may read Bob's comment before Alice's message.

Even if it boosts availability, a relaxed consistency model can lead to conflicts when concurrent operations are invoked on different replicas. Traditionally, conflicts are reconciled manually or left to the application to decide on the order (all concurrent versions are retained and exposed) [192]. This process is very costly and subject to errors which necessitates a systematic way instead. Conflict-free Replicated DataTypes [181, 97, 98] (CRDTs) are mathematical abstractions that are proven to be conflict-free and easy to use, and they are recently being adopted by leading industry like Facebook, Tom Tom, Riak DB, etc.. In the rest of this section, we introduce CRDTs and present some of the recent advanced models.

4.3.2 Conflict-free Replicated Datatypes

Conflict-free Replicated Datatypes (CRDTs) [181, 97, 98] are data abstractions (e.g, counters,sets, maps, etc), that are usually replicated over (often loosely-connected) network of nodes, and are mathematically formalized to guarantee that replicas converge to the same state, provided they eventually apply the same updates. The assumption is that state updates should be commutative, or designed to be so. This ensures that applying concurrent updates on different nodes is independent from their application order. For instance, applying "Add A" and "Add B" to a *set* will eventually lead to both elements A and B in the two replicas. Whereas, concurrent non-commutative operations "Add A" and "Remove A" may lead to two different replicas (one is empty and the other has element A).

There are two main approaches for CRDTs: operation-based (a.k.a., op-based) and state-based. The former is based on disseminating operations whereas the latter propagates a state that results from locally applied operations. In the systems where the message dissemination layer guarantees Reliable Causal Broadcast (RCB), op-based CRDTs are desired as they allow for simpler implementations, concise replica state, and smaller messages. On the other hand, state-based CRDTs are more complex to design, but can handle duplicates and out-of-order delivery of messages without breaking causality, and thus are favored in hostile networks.

In the following, we elaborate on these models and their optimizations focusing on the Add-wins Set (AWSet) datatype (in which a concurrent add dominates a remove). We opt for AWSet being a very common datatype that has causality semantics, which helps explaining the different ordering guarantees in CRDT models. More formally, in an AWSet that retains add and rmv operations together with their timestamps t, the concurrent semantics can be defined by a read operation that returns the following elements:

$$\{v \mid (t, [\mathsf{add}, v]) \in s \wedge \forall (t', [(\mathsf{add} \mid \mathsf{rmv}), v]) \in s \cdot t \not< t'\} \tag{4.1}$$

Op-based CRDTs. In op-based CRDTs [181], each replica maintains a local state that is type-dependent. A replica is subject to clients' operations, i.e., *query* and *update*, that are executed locally as soon as they arrive. While query operations read the local (maybe stale) state without modifying it, updates often modify the state and generate some message to be propagated to other replicas through a reliable causal broadcast (RCB) middleware. The RCB handles the exactly-once dissemination across replicas, mainly through an API composed of causal broadcast function cbcast and causal deliver function cdeliver.

Σ	: State type, σ_i is an instance
$\mathsf{prepare}_i(o, \sigma_i)$	: Prepares a message m given an operation o
$\mathsf{effect}_i(m, \sigma_i)$	: Applies a *prepared* message m on a state
$\mathsf{val}_i(q, \sigma_i)$	: Read-only evaluation of query q on a state

Figure 4.15: The general composition of an op-based CRDT.

To explain the process further, we consider the general structure of an op-based CRDT in Figure 4.15, and convey Algorithm 1 in Figure 4.16 that depicts the interplay between the RCB middleware and the CRDT. In particular, when an update operation o is issued at some node i having state σ_i, the function $\mathsf{prepare}_i(o, \sigma_i)$ produces a message m that includes some ordering meta-data in addition to the operation. This message m is then broadcast by calling $\mathsf{cbcast}_i(m)$, and is delivered via $\mathsf{cdeliver}_j(m)$ at each destination node j (including i itself). cdeliver triggers $\mathsf{effect}_j(m, \sigma_j)$ that

```
state:
    $\sigma_i \in \Sigma$
on operation$_i(o)$:
    $m :=$ prepare$_i(o, \sigma_i)$
    cbcast$_i(m)$
on cdeliver$_i(m)$:
    $\sigma_i :=$ effect$_i(m, \sigma_i)$
on query$_i()$:
    val$_i(\sigma_i)$
```

Algorithm 1: CRDT & RCB

1

```
state:
    $\sigma_i = (s, \pi) \in \mathcal{P}(O) \times (O, \leq)$
on operation$_i(o)$:
    tcbcast$_i(o)$
on tcdeliver$_i(m, t)$:
    $\pi_i :=$ effect$_i(m, t, \pi_i)$
on tcstable$_i(t)$:
    $\sigma_i :=$ prune$_i(\pi_i, s, t)$
on query$_i()$:
    val$_i(\sigma_i)$
```

Algorithm 2: Pure CRDT & TRCB

Figure 4.16: Distributed algorithms for node *i* showing the interplay of classical and pure op-based CRDTs given a standard versus tagged reliable causal broadcast middlewares, respectively.

$$\begin{array}{rcl}
\Sigma = \mathbb{N} \times \mathscr{P}(I \times \mathbb{N} \times V) & & \sigma_i^0 = (0, \{\}) \\
\mathsf{prepare}_i([\mathsf{add}, v], (n, s)) & = & [\mathsf{add}, v, i, n+1] \\
\mathsf{effect}_i([\mathsf{add}, v, i', n'], (n, s)) & = & (n' \wedge (i = i') \vee n, s \cup \{(v, i', n')\}) \\
\mathsf{prepare}_i[\mathsf{rmv}, v], (n, s)) & = & [\mathsf{rmv}, \{(v', i', n') \in s \mid v' = v\}] \\
\mathsf{effect}_i([\mathsf{rmv}, r], (n, s)) & = & (n, s \setminus r) \\
\mathsf{val}_i(n, s) & = & \{v \mid (v, i', n') \in s\}
\end{array}$$

Figure 4.17: Operation-based Add-Wins Set CRDT, at node *i*.

returns a new replica state σ_j'. Finally, a query operation q is issued, $\mathsf{val}_i(q, \sigma_i)$ is invoked, and no corresponding broadcast occurs.

Example.

We further exemplify op-based CRDTs though an "Add-wins Set" (AWSet) CRDT design depicted in Figure 4.17. The state Σ is composed of a local sequence number $n \in \mathbb{N}$ and a set of values in V together with some meta-data in $I \times \mathbb{N}$ that is used to guarantee the causality information of the datatype. The prepare of an add operation produces a tuple, to be disseminate by RCB, composed of the element to be added together with the ID and the incremented sequence number of the local node *i*. The effect function is invoked on RCB delivery and leads to adding this tuple to the state and incrementing the sequence number only if the prepare was local at *i*. To the contrary, preparing a rmv returns all the tuples containing the removed item, which allows the corresponding effect to remove all these tuples. Finally, the val function returns all the elements in the set.

Pure Op-based CRDTs. Op-based CRDTs can be optimized to reduce the dissemination and storage overhead. Indeed, since op-based CRDTs assume the presence of and RCB, one can take advantage of its time abstractions, i.e., often implemented

$$\begin{array}{rcl}
\Sigma = \mathscr{P}(O) \times T \hookrightarrow O & & \sigma_i^0 = (s_i, \pi_i)^0 = (\{\}, \{\}) \\
\mathsf{effect}_i(o,t,s,\pi) & = & (s \setminus \{[\mathsf{add}, v] \mid o = [\mathsf{rmv}, v]\}, \\
 & & \pi \setminus \{(t', [\mathsf{add}, v]) \mid o = [\mathsf{rmv}, v] \wedge t' < t\} \cup \\
 & & \{(t,o) \mid o = [\mathsf{add}, v] \wedge (\_, o) \notin \pi \wedge o \notin s\}) \\
\mathsf{val}_i(s,\pi) & = & \{v \mid [\mathsf{add}, v] \in s \vee (t, [\mathsf{add}, v]) \in \pi\} \\
\mathsf{prune}_i(\pi, s, \tau) & = & (s \cup \{o\}, \pi \setminus \{(t,o)\}) \wedge t \leq \tau
\end{array}$$

Figure 4.18: Pure operation-based Add-Wins Set CRDT, at node *i*.

via version vectors (VV), to guarantee causal delivery and thus avoid disseminating the meta-data produced by prepare. Consequently, this leads to disseminating the "pure" operations and possible arguments which makes the prepare useless, hence the name Pure op-based CRDTs [98, 183]. In addition, this "pure" mind-set leads to having a standard state for all datatypes: a partially order log: Polog. However, this will lead to storing the VV in the state which can be very expensive when the number of replicas in the systems is high. Fortunately, and contrary to the classical op-based approach, the time notion of VV is useful to transform the Polog into a sequential log which eventually helps to prune the—no longer needed—VVs.

Tagged Reliable Causal Broadcast (TRCB). To achieve the above optimizations, we consider an extended RCB, called Tagged RCB (TRCB), that provides two functionalities through the API functions: tcdeliver and tcstable [98, 183]. The former is a equivalent to the standard RCB cdeliver presented above with a simple extension to the API by exposing the VVs (used internally by the RCB) to the node upon delivery, which can be appended to the operation by the recipient. On the other hand, tcstable is a new function that returns a timestamp τ indicating that all operations with timestamp $t \leq \tau$ are *stable*: have been delivered on all nodes. The essence is that no concurrent operations to the stable operations in the Polog are expected to be delivered, and hence, the corresponding VVs in the Polog can be pruned without affecting the datatype semantics.

Given the TRCB, depicted in Algorithm 2 of Figure 4.16, the design of Pure CRDTs differs from the classical ones in different aspects. First, the state is common to all datatypes, and is represented by a set of stable operations and a Polog. Second, prepare has no role anymore as the operations and its arguments can be immediately disseminated though the TRCB. Third, the significant change is with the effect function that discards datatype-specific "redundant" operations before adding to the Polog (e.g., a duplicate operation). However, to the contrary of classical op-based CRDTs, only one effect function is required per datatype. Finally, prune function is required to move stable operations (triggered via the TRCB's tcstable) to the sequential log after pruning the VVs.

Example. Using the same example of the AWSet, we provide the Pure CRDT version in Figure 4.18. The state is composed of a set of sequential operations s and Polog π: a map from timestamps to operations. The prepare does not exist due the reasons mentioned above, and hence the operation and its arguments are sent to the destination. Once a rmv operation is delivered, the effect deletes the corresponding operation from s and those in the causal past of $\mathsf{rmv} \in \pi$. (Remember that all operations in s are in the causal past of delivered operations.) Finally, effect only adds an add to π if the element is not in s or π. Once the tcstable triggers prune, given a stable timestamp τ, all operations with timestamp $t \leq \tau$ become stable; and are thus removed by prune from π and added to s without the (now useless) timestamp.

Additional pedantic details. A catalog of many op-based CRDT specifications like counters, sets, registers, maps, etc., can be found in [183]. There are also several optimizations that are beyond the scope of this book. For instance, the pure op-based specifications can be generalized further to have a common framework for all CRDTs in such a way the user only needs to define simple datatype-specific rules to truncate the Polog. In addition, one can go deeper and optimize each datatype aside. An example is to replace the stable state with a classical datatype instead of retaining the set of operations. On the other hand, datatypes that are natively commutative can be easier to implement as classical op-based CRDTs. Finally, we avoid presenting the details of the TRCB for presentation purposes. The reader can refer to [98, 183] for these details.

State-based CRDTs. While op-based CRDTs are based on the dissemination of operations that are executed by every replica, a "state" is disseminated in the state-based CRDTs [181]. A received state is incorporated with the local state via a *merge* function that, deterministically, reconciles any existing conflicts.

Σ	: State defined as *join semi-lattice*, σ_i is an instance
$Mutators$	: mutating operations that inflate the state.
$s \sqcup s'$	: LUB to merge states s and s'
$\mathsf{val}_i(q, \sigma_i)$	: Read-only evaluation of query q on a state

Figure 4.19: The general composition of a state-based CRDT.

As depicted in Figure 4.19, a state-based CRDT consists of a state, mutators, join, and query functions. The state Σ is designed as a join-semilattice [193]: a set with a *partial order*, and a binary *join* operation $\sqcup$ that returns the *least upper bound* (LUB) of two elements in S, and is always designed to be commutative, associative, and idempotent. On the other hand, mutators are defined as *inflation*: for any mutator m and state X, $X \sqsubseteq m(X)$. This guarantees that the state never diminishes, and thus, each subsequent state subsumes the previous state when joined elsewhere. Finally, the query operation leaves no changes on the state. Note that the specification of all these functions, and the state, are datatype-specific.

Anti-entropy protocol. To the contrary of op-based CRDTs that assume the presence of RCB, state-based CRDTs can ensure eventual convergence using a simple *anti-entropy* protocol, as in Figure 4.20, that periodically ships the entire local state to other replicas. Each replica merges the received state with its local state using the *join* operation. (The algorithm in Figure 4.20 can be more sophisticated to include retransmissions, routing, or gossiping, but we keep it simple for presentation purposes.).

```
inputs:
    $n_i \in \mathcal{P}(\mathbb{I})$, set of neighbors
durable state:
    $X_i := \bot \in S$, CRDT state
on receive$_{j,i}(Y)$
    $X_i' = X_i \sqcup Y$
on operation$_i(m)$
    $X_i' = m(X_i)$
periodically // ship state
    $j = \mathsf{random}(n_i)$
    send$_{i,j}(X_i)$
```

1

Figure 4.20: A basic anti-entropy algorithm for state-based CRDTs.

$$
\begin{aligned}
\Sigma &= \mathscr{P}(\mathbb{I}\times\mathbb{N}\times E)\times\mathscr{P}(\mathbb{I}\times\mathbb{N})\\
\sigma_i^0 &= (\{\},\{\})\\
\mathsf{add}_i(e,(s,t)) &= (s\cup\{(i,n+1,e)\},t)\\
&\quad \text{with } n=\max(\{k\mid(i,k,\_)\in s\})\\
\mathsf{rmv}_i(e,(s,t)) &= (s,t\cup\{(j,n)\mid(j,n,e)\in s\})\\
\mathsf{val}_i((s,t)) &= \{e\mid(j,n,e)\in s\wedge(j,n)\notin t\}\\
(s,t)\sqcup(s',t') &= (s\cup s',t\cup t')
\end{aligned}
$$

Figure 4.21: State-based AWSet CRDT, at node i.

Example. Again, we exemplify on state-based CRDTs via Figure 4.21 that depicts the design of AWSet. The state Σ is composed of two sets. One set is for the addition of elements with unique tags defined by unique ID and sequence number, and another tombstones set that serves for collecting the removed tags. This design is crucial to achieve the semi-lattice inflation properties subject to mutators: add and rmv. The former adds the new element to the addition set together with a new tag leaving the tombstones set intact. An element is removed by rmv through adding its unique tag to the tombstones set. Notice that the element must not be removed from the addition set which violates inflation. Given these specifications, the query function val will simply return all the added elements that do not have corresponding tags in the tombstones set. Finally, the merge function $\sqcup$ joins any two (disseminated or not) states by simply computing the union of the sets, thus respecting the semi-lattice properties.

Delta-state CRDTs. Despite the simplicity and robustness of state-based CRDTs, the dissemination overhead is high as the entire state is always propagated even with small local state updates. Delta-state CRDTs [97, 182] are state-based CRDT variants that allow to "isolate" the recent updates on a state and ship the corresponding *delta*, i.e., a state in the semi-lattice corresponding only to the updates, and shipped to be merged remotely. The trick is to find new delta-mutators (a.k.a., δ-mutators) m^δ that return deltas instead of entire states (and again, these are datatype specific). Given a state X, the relation between mutators m in state-based CRDTs and m^δ is as follows:

$$X' = m(X) = X \sqcup m^\delta(X) \tag{4.2}$$

$$\begin{array}{rcl}
\Sigma & = & \mathscr{P}(\mathbb{I} \times \mathbb{N} \times E) \times \mathscr{P}(\mathbb{I} \times \mathbb{N}) \\
\sigma_i^0 & = & (\{\}, \{\}) \\
\mathsf{add}_i^\delta(e, (s,t)) & = & (\{(i, n+1, e)\}, \{\}) \\
 & & \text{with } n = \max(\{k \mid (i,k,\_) \in s\}) \\
\mathsf{rmv}_i^\delta(e, (s,t)) & = & (\{\}, \{(j,n) \mid (j,n,e) \in s\}) \\
\mathsf{val}_i((s,t)) & = & \{e \mid (j,n,e) \in s \wedge (j,n) \notin t\} \\
(s,t) \sqcup (s',t') & = & (s \cup s', t \cup t')
\end{array}$$

Figure 4.22: Delta-state AWSet CRDT, at node *i*.

This represents the main change in the design of state-based CRDTs in Figure 4.19.

Example. Considering the AWSet example, the only difference between the delta CRDT version in Figure 4.22 and its state-based counterpart in Figure 4.21 is with mutators. One can simply notice that add^δ returns the recent update that represents the added element whereas add returns the entire state. A similar logic holds for the difference between rmv^δ and rmv. As for the $\sqcup$ and val, their design is the same in both versions; however, notice that the propagated and merged message is a delta-state in Figure 4.22 rather than a whole state. Indeed, although a delta-mutator returns a single delta, it more practical to join deltas locally and ship them in groups (which must not affect the $\sqcup$ in any case) as we explain next.

Causal anti-entropy protocol.. This delta CRDT optimization comes at a cost: it is not longer safe to blindly merge received (delta) states when the datatype requires causal semantics. Indeed, state-based CRDTs implicitly ensure per-object causal consistency since the state itself retains all the causality information, whereas a delta state includes the tags of individual changes without any memory about the causal past. This requires a little more sophisticated anti-entropy protocol that enforces causal delivery on received deltas and supports coarse-grained shipping of delta batches, called "delta-intervals". A delta-interval $\Delta_i^{a,b}$ is a group of consecutive deltas correspOnding to a sequence of all the local delta mutations from a through $b-1$, and merged together via $\sqcup$ before shipping:

$$\Delta_i^{a,b} = \bigsqcup\{d_i^k \mid a \leq k < b\} \tag{4.3}$$

Given this, an anti-entropy algorithm can guarantee causal order if it respects the "causal delta-merging condition": $X_i \sqsupseteq X_j^a$. This means that a receiving replica can only join a remote delta-interval if it has already seen (and merged) all the causally preceding deltas for the same sender. The algorithm in Figure 4.23 is a basic anti-entropy protocol that satisfies the causal delta-merging property. (We discarded many optimization to focus on the core concept.) In addition to the state, a node retains a sequence number that, together with the acknowledgments map, helps the node to identify the missing deltas to be sent to a destination. In addition, the delta-interval D serves to batch deltas locally before sending them periodically. Once an operation is received from a client, a corresponding delta is returned by m^δ, which is then merged to the local state and joined to D for later dissemination to a random node. Once a delta interval is received, it gets merged to the local state as well as the local delta-interval buffer—to be sent to other nodes. Finally, deltas that have been sent to all nodes are garbage-collected from D.

```
inputs:
    $n_i \in \mathcal{P}(\mathbb{I})$, set of neighbors
durable state:
    $X_i := \bot \in S$, CRDT state
    $c_i := 0 \in \mathbb{N}$, sequence number
volatile state:
    $D_i := \{\} \in \mathbb{N} \hookrightarrow S$, sequence of $\delta$s
    $A_i := \{\} \in \mathbb{I} \hookrightarrow \mathbb{N}$, ack map
on $\mathsf{operation}_i(m^\delta)$
    $d = m^\delta(X_i)$
    $X_i' = X_i \sqcup d$
    $D_i' = D_i\{c_i \mapsto d\}$
    $c_i' = c_i + 1$
on $\mathsf{receive}_{j,i}(\mathsf{delta}, d, n)$
    if $d \not\sqsubseteq X_i$ then
        $X_i' = X_i \sqcup d$
        $D_i' = D_i\{c_i \mapsto d\}$
        $c_i' = c_i + 1$
    $\mathsf{send}_{i,j}(\mathsf{ack}, n)$
on $\mathsf{receive}_{j,i}(\mathsf{ack}, n)$
    $A_i' = A_i\{j \mapsto \max(A_i(j), n)\}$
periodically // ship delta-interval
    $j = \mathsf{random}(n_i)$
    $d = \bigsqcup\{D_i(l) \mid A_i(j) \leq l < c_i\}$
    $\mathsf{send}_{i,j}(\mathsf{delta}, d, c_i)$
periodically // garbage collect $\delta$s
    $l = \min\{n \mid (\_, n) \in A_i\}$
    $D_i' = \{(n, d) \in D_i \mid n \geq l\}$
```

Figure 4.23: Basic causal anti-entropy protocol satisfying the delta-merging condition.

Additional pedantic details. A catalog of many delta-state CRDT specifications like counters, sets, registers, maps, etc., can be found in [97, 182]. There are also several optimizations that are beyond the scope of this book. For instance, the tags in the tombstone set can be compressed further in a single version vector and few tags. This helps generalizing the specifications to use a common causality abstraction per all datatypes [182]. Furthermore, the causal anti-entropy protocol can consider other conditions to improve performance, e.g., through considering transitive propagation of deltas or sending a complete state once a delta does not help, e.g., a node was unavailable for a long time. In this particular case, other useful alternatives to define deltas by *join decomposition* can be found in [194].

4.3.3 A case study: dataClay distributed platform

We now present a case study to demonstrate the practical use of CRDTs in a real distributed system: dataClay distributed platform [184, 185]. The aim is to give the reader an applied example of CRDTs showing how they can make the developers life easier. For that purpose, we try to be direct and simple to help the reader getting started.

dataClay. A distributed platform aimed at storing, either persistently or in memory, Java and Python objects [184, 185]. This platform enables, on the one hand, to store objects as in an object-oriented database and, on the other hand, to build applications where objects are distributed among different nodes, while still being accessible from any of the nodes where the application runs. Furthermore, dataClay enables several applications to share the same objects as part of their data set.

$$\begin{aligned}
\Sigma &= I \hookrightarrow \mathbb{N} \\
\sigma_i^0 &= \{\} \\
\mathsf{inc}_i(m) &= m\{i \mapsto m(i)+1\} \\
\mathsf{val}_i(m) &= \sum_{r \in \mathbf{dom}(m)} m(r) \\
m \sqcup m' &= \mathbf{max}(m, m')
\end{aligned}$$

Figure 4.24: State based GCounter CRDT, on replica *i*.

dataClay has three interesting properties. The first is that it stores the class methods in addition to the data. This functionality has several implications that help application developers use the data in this platform: (1) data can only be accessed using the class methods (no direct field modifications) and thus class developers can take care, for instance, of integrity constraints that will be fulfilled by all applications using the objects; and (2) methods can be executed over the objects inside the platform, without having to move the data to the application. The second property is that objects in dataClay are not flat; and they can rather be composed of other objects, or language basic types, like in any object-oriented language. Finally, dataClay enables objects to be replicated to several nodes managed by the platform in order to increase tolerance to faults and/or execution performance by exploiting parallelism.

The case for CRDTs. Despite fully replicating the data (and class definitions) to improve tolerance to faults, dataClay does not natively implement any synchronization scheme between replicas since some applications (or modules of an application) cannot afford paying the synchronization price [195, 186]. Consequently, to provide this flexibility, dataClay tries to offer mechanisms for class developers to implement the consistency model their objects may need. Nevertheless, building such mechanisms is always tedious and, most importantly, synchronization implies a performance penalty and lack of scalability [189]. Here is where CRDTs come into play to provide a relaxed consistency and seamless plug-and-play conflict resolution for the replicated objects across the platform. Furthermore, given that code is part of the replicated data, the class developers can implement CRDTs and dataClay itself will guarantee that the update rules will be followed regardless of the application using them.

Using CRDTs. For replicating objects, dataClay can provide the developer with a library for CRDTs to be used in the classes and maybe through composing objects. Given that dataClay's system model is a graph-like Peer-to-Peer system, it is more desirable to use the state-based CRDT model since no RCB middleware is required. By using CRDTs, any application can modify the data objects without prior synchronization with other replicas or applications. Once the changes are propagated, CRDTs can eventually converge to the same value. In order to show how CRDTs are mapped to dataClay, we provide a simple example on a Grow-only Counter

(GCounter) CRDT [181, 97]. We choose the counter being a simple example and at the same time shows how a semi-lattice can be different from the AWSet discussed before.

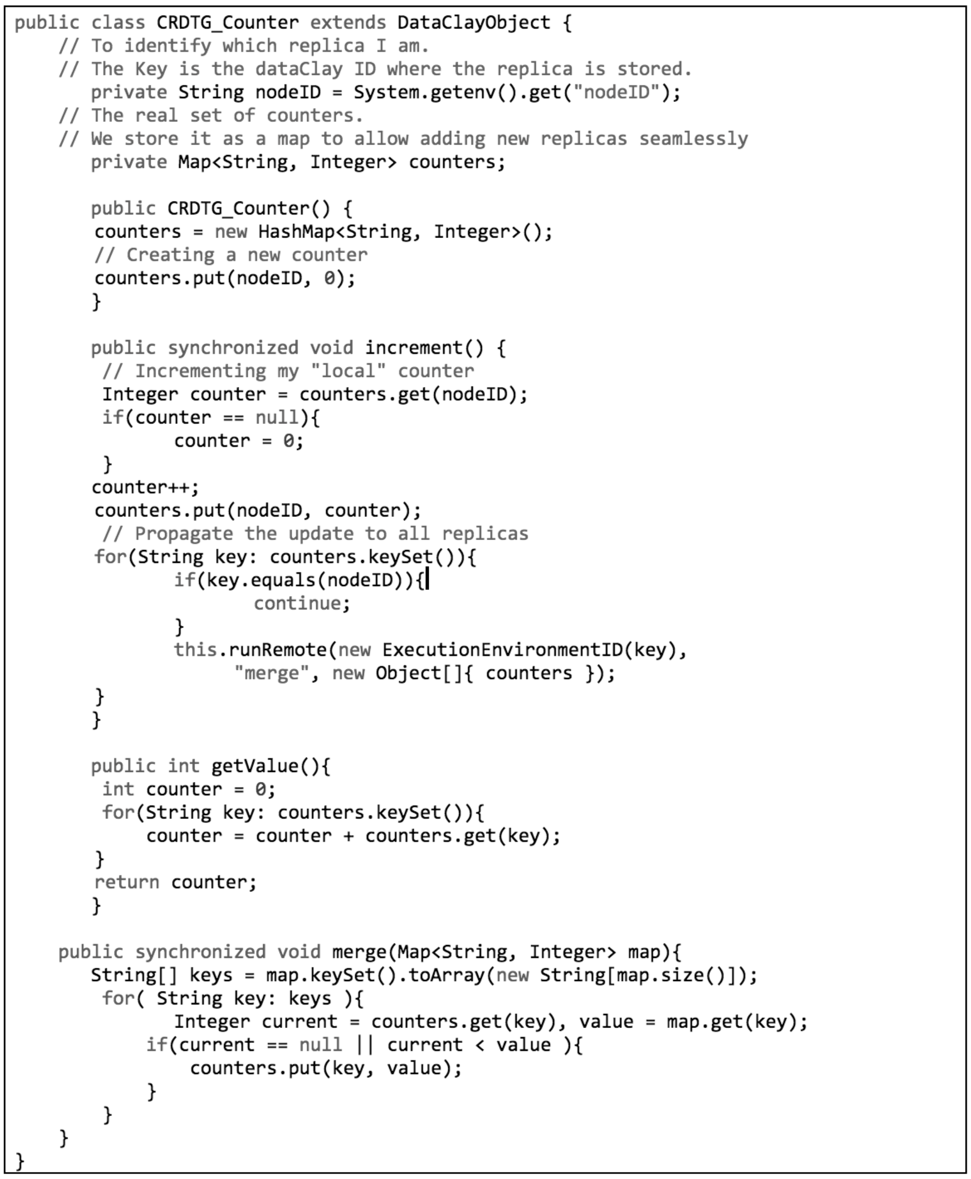
The GCounter specification is conveyed in Figure 4.24. In this design, the state Σ is defined as a map from node IDs to a natural number corresponding to the increments done locally. As inc mutator shows, a node can only increment its own key, whereas the val query function returns the sum of all keys from all nodes. Once a (whole) state is propagated, the merge is done through taking the maximum counter corresponding to each key. For instance, in a system of three nodes, the following two GCounters are merged as follows: $(1,4,5) \sqcup (3,4,2) = (3,4,5)$. In the *Java* implementation of the GCounter presented in Figure 4.25, the state is coded as a *hash map* to enable the addition of new replicas on the fly, without any kind of synchronization and/or notification as part of the CRDT. In this code, the increment method pushes the new version of the hash map to all existing replicas to have the last up-to-date version and recover any potentially missed update from another node. We leave the details of the code as an exercise to the reader.

Class deployment. As mentioned above, dataClay also replicates the class definitions to be used by the applications across the systems and to allow method invocations close to the data. For this purpose, once a class is updated, the different versions must be coordinated to avoid conflicts in the semantics of the corresponding class instances. In order to allow for these updates in a loosely coordinated fashion, the classes can be designed as a Set CRDT associated with the class version. When a class update is made somewhere by any developer, the changes are deployed everywhere in the system, but they cannot be used until all replicas in the system see the new change. This concept is similar to the *causal stability* feature provided by the Tagged RCB presented before. In particular, although not all nodes detect causal stability of a version at the same time, once any node detects this, it is safe to start using that version. The reason is that causal stability ensures that the version has been delivered by all nodes in the system, and thus, the new class updates can be fetched to be used in the future. Notice that we are talking about class deployment here, but the designer must consider the compatibility between the old and the new versions, e.g., if some class instances already exist.

4.3.4 Conclusions and Future Directions

CRDTs make using replicated data less cumbersome to developers and correct being mathematically designed abstractions. However, they can only be useful when the application semantics allow for stale reads and favor immediate writes. CRDTs exist for many datatype variants of counters, sets, maps, registers, graphs, etc. They can however be extended to other types as long as operations are commutative or can be made so.

This section presented two main variants for CRDTs and their important optimizations. Some of the tradeoffs are understood, while others require future empirical investigation. Op-based CRDT designs are more intuitive to design and can be used once a reliable causal middleware is available. If it is possible to extend the middleware API, once can use pure op-based CRDTs to reduce the overhead of



```
public class CRDTG_Counter extends DataClayObject {
    // To identify which replica I am.
    // The Key is the dataClay ID where the replica is stored.
       private String nodeID = System.getenv().get("nodeID");
    // The real set of counters.
    // We store it as a map to allow adding new replicas seamlessly
       private Map<String, Integer> counters;

       public CRDTG_Counter() {
       counters = new HashMap<String, Integer>();
       // Creating a new counter
       counters.put(nodeID, 0);
       }

       public synchronized void increment() {
        // Incrementing my "local" counter
        Integer counter = counters.get(nodeID);
        if(counter == null){
              counter = 0;
        }
       counter++;
       counters.put(nodeID, counter);
        // Propagate the update to all replicas
       for(String key: counters.keySet()){
              if(key.equals(nodeID)){
                     continue;
              }
              this.runRemote(new ExecutionEnvironmentID(key),
                   "merge", new Object[]{ counters });
       }
       }

       public int getValue(){
        int counter = 0;
        for(String key: counters.keySet()){
            counter = counter + counters.get(key);
       }
       return counter;
       }

    public synchronized void merge(Map<String, Integer> map){
       String[] keys = map.keySet().toArray(new String[map.size()]);
        for( String key: keys ){
              Integer current = counters.get(key), value = map.get(key);
            if(current == null || current < value ){
                counters.put(key, value);
            }
        }
    }
}
```

Figure 4.25: An implementation for GCounter CRDT in dataClay.

dissemination and storage. On the other hand, state-based CRDTs are more tolerant in hostile networks and gossip-like systems being natively idempotent: data can arrive though different nodes and get merged safely. Despite being easy to use in practice, state-based CRDTs can be expensive on dissemination when the state is not small. Consequently, it is recommended to use the delta-state CRDT alternative that significantly reduces the dissemination cost if a convenient causal anti-entropy protocol can be deployed. Furthermore, hybrid models of these variants can have tradeoff properties, and are interesting to study in the future work. Finally, it would be promising to investigate the feasibility of CRDTs in other system models and research areas like Edge Computing or Blockchain.

4.4 Summary

Given current projections for data growth and the needs for increased data processing, ultrascale systems face significant challenges for keeping up both with data storage and access. This chapter has examined approached to tackle the efficiency of data storage and access at three levels: the storage level via an efficient key-value store, the workflow-level via a locality aware workflow management systems, and the algorithmic-level via the use of conflict-free replicated data types.

Midterm challenges related to data management in Ultrascale system will require more research in topics related to:

HPC and data analysis: Understand and realize the close relationship between HPC and data analysis in the scientific computing area and advances in both are necessary for next-generation scientific breakthroughs. To achieve the desired unification, the solutions adopted should also be portable and extensible to future Ultrascale systems. These systems are envisioned as parallel and distributed computing systems, reaching two to three orders of magnitude larger than today's systems.

Embrace and cope with new storage device technologies: The appearance of new storage device technologies carry a lot of potential for addressing issues in these areas, but also introduce numerous challenges and will imply changes on the way data is organized, handled, and processed, throughout the storage and data management stack.

Shift from performance to efficiency: Instead of only or mostly considering absolute performance as the driving force for new solutions, we should shift our interest to considering the efficiency at which infrastructures are operating. This is becoming more important as the size at which future infrastructures are required to operate continues to scale with application requirements.

To face those challenges, we will need to enforce the convergence of HPC, Ultrascale and Big Data worlds. Storage, interconnection networks and data management in both HPC and Cloud needs to cope with technology trends and evolving application requirements, while hiding the increasing complexity at the architectural, systems software, and application levels. Future work needs to examine these challenges under the prism of both HPC and Cloud approaches and to consider solutions that break away from current boundaries. Moreover, future applications will need more sophisticated interfaces for addressing the challenges of future Ultrascale computing systems. These novel interfaces should be able to abstract architectural and operational issues from requirements for both storage and data. This will allow applications and services to easier manipulate storage and data, while providing the system with flexibility to optimize operation over a complex set of architectural and technological constraints.