Fine-grained data caching approaches to speedup a distributed RDataFrame analysis

Vincenzo Eduardo Padulano[1,2,*], *Enric* Tejedor Saavedra[2,**], and *Pedro* Alonso-Jordá[1,***]

[1]Universitat Politècnica de València. Camino de Vera, s/n 46022 Valencia (Spain)
[2]CERN. Esplanade des Particules 1 1211 Geneva 23 (Switzerland)

Abstract. Thanks to its RDataFrame interface, ROOT now supports the execution of the same physics analysis code both on a single machine and on a cluster of distributed resources. In the latter scenario, it is common to read the input ROOT datasets over the network from remote storage systems, which often increases the time it takes for physicists to obtain their results. Storing the remote files much closer to where the computations will run can bring latency and execution time down. Such a solution can be improved further by caching only the actual portion of the dataset that will be processed on each machine in the cluster, reusing it in subsequent executions on the same input data. This paper shows the benefits of applying different means of caching input data in a distributed ROOT RDataFrame analysis. Two such mechanisms will be applied to this kind of workflow with different configurations, namely caching on the same nodes that process data or caching on a separate server.

1 Introduction

The high amount of data collected by the LHC experiments has made distributed computing a staple in High Energy Physics (HEP) data processing workflows for a long time, with the WLCG [1] being the prime example of efforts in that direction. The collider is scheduled for a major upgrade in the next years that will allow for more accurate measurements of physics processes. The upgraded machine called HL-LHC [2] is scheduled to begin delivering data in 2027 and will bring forth new challenges in data storage and computing. It is foreseen that it will generate roughly thirty times more data than the LHC has produced so far. Given the available future budget estimations and expected technological evolutions [3], experiments at the LHC and the larger HEP community would benefit from improvements on the software. It will be crucial to make the most out of current and future architectures. In this regard, distributed computing will need to be revisited with new approaches, algorithms and frameworks. For example, letting the user interactively explore their dataset even as it grows larger and larger will be a requirement in many physics analysis groups. Services such as SWAN [4] try to solve that need, providing a modern interactive interface for analysis through Jupyter notebooks and the possibility to run on distributed cluster resources on demand.

In this context, ROOT RDataFrame [5] provides a high level programming model to define analyses in terms of a computation graph that can be parallelized to run both on a

[*]e-mail: vincenzo.eduardo.padulano@cern.ch
[**]e-mail: enric.tejedor.saavedra@cern.ch
[***]e-mail: palonso@upv.es

multi core machine and a set of distributed resources. This second option enables for example submitting RDataFrame applications through multiple tasks in a distributed computing framework like Apache Spark [6]. This combination has already been tested in real world scenarios [7], showing good potential in lowering user analysis runtime and providing a first example of interactivity in this kind of workflow.

Nevertheless, it should not be forgotten that in physics analyses, whether running on a user laptop or on a cluster of machines, the dataset size can range from a few GBs to multiple TBs. These datasets are typically stored on some remote storage systems. For example at CERN the EOS open storage technology [8] is widely used for this purpose. Data has to be read remotely during analysis runtime, leading to performance penalties that can become real bottlenecks depending also on the distance between the storage facility and the computing nodes.

One strategy that can play a key role in this sense is to cache the input data as close as possible to the place of processing. Over the years, different groups have presented various approaches to this issue, many revolving around the usage of the well established XRootD framework [9]. In particular, XRootD provides both a communication protocol for network transactions involving reading and writing data and an extensible set of data management plugins. Notably, a disk-based plugin was developed to enable caching locally on the computing cluster [10]. This mechanism, also known as XCache, has been used in many applications including data management systems of CERN experiments like ATLAS [11].

Similar caching approaches could be greatly beneficial in RDataFrame analyses as well, especially if it is considered to distribute the workload over a set of distributed resources that could offer from tens to thousands of cores. It is actually common that a physicist needs to explore the features of a dataset over multiple subsequent application runs, each time modifying the code slightly to account for different parameters or desired statistics. In such use cases caching would give a substantial runtime reduction to those application runs that find an already populated cache. For example, one solution investigated in this work is XRootD caching integration in an RDataFrame application. One possible downside to such system is that XRootD is not aware of the internals of the ROOT data format. For that reason, it is possible that a new caching mechanism native to ROOT could make better use of a tighter integration between the programming model and the local storage of the computing nodes. To this end, the TFilePrefetch [12] class provides a lightweight agent running along the main event loop and ROOT data as it is being read from the remote storage system. Despite its potential benefits over XRootD, it is still an experimental component of the ROOT framework and needs thorough validation.

In this work, the two caching mechanisms mentioned above are tested and compared in an RDataFrame application, to better understand benefits and drawbacks of each one.

2 Programming model of a distributed RDataFrame analysis

An RDataFrame application is expressed in a declarative way: the user specifies the transformations to be applied to the input dataset (e.g. filtering entries, adding new columns) plus the actions to be performed in order to obtain the final results (e.g. histograms), thus constructing a computation graph. Data parallelism can be employed to parallelize the execution of such graphs: the same operations are applied to multiple fragments of the input dataset, before the partial results are merged into the final one. This map-reduce pattern is implemented by RDataFrame via threads in a local machine, but also by leveraging distributed computing frameworks such as Spark.

This distributed functionality is provided through a Python interface to RDataFrame [13]. It wraps the computation graph in a map reduce pattern and then splits logically the input

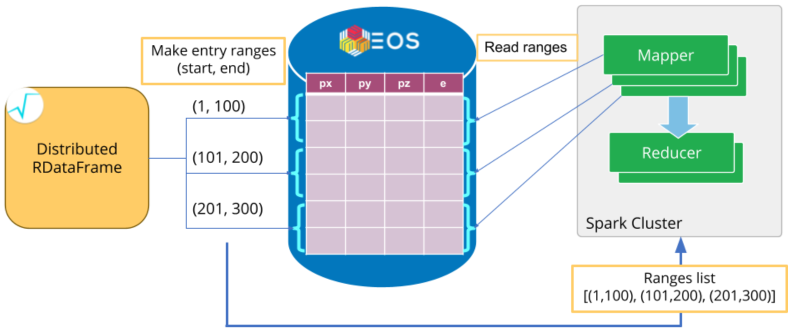

Figure 1. Creation of tasks in the distributed RDataFrame module. The input dataset is split in logical ranges of entries. The list with all the ranges is sent to the Spark scheduler, which creates one map task for each range. Each mapper reads and processes only the entries in the range it got assigned to. The partial results coming from the mappers are aggregated in the reduce phase and finally sent back to the user.

dataset into multiple ranges of entries (see Figure 1). The workload is thus divided into tasks, each one made of the map function plus one logical data range, that will be sent to the computing nodes. In the reduce phase, all the finished tasks are aggregated into the final results, which are finally streamed back from the cluster to the user application.

With RDataFrame it is possible to read and process datasets stored in the ROOT file format, either from the local filesystem or remotely. More specifically, the I/O implementation of this columnar format enables independent reading and writing of the dataset in smaller chunks called *clusters*. A collection of *clusters* then represents a column of the dataset, also called *TTree* or simply tree, that in turn is saved into the actual ROOT file. When reading a remote ROOT file, a *cluster* is the smallest data volume that can be streamed independently from the rest of the file, that is without triggering extra read operations from the same dataset in memory and over the network. This feature can be exploited in the context of caching the files. Caching only the portions of dataset needed for processing on each computing node will directly translate into less network traffic. This holds true for both dimensions in which the dataset can be partitioned: along columns (only a subset of features for every event) or along *clusters* (ranges of events).

3 Caching mechanisms

In a physicist's exploratory analysis workflow, it is common to rerun an application multiple times on the same input data, with slightly modified code. This opens the door to caching the dataset (or better yet the portion of it which is actually processed) during the first run of the user application. This will speed up subsequent runs where the computing nodes can read data from the cache rather than from the remote storage. The caching mechanism should be as transparent as possible for the user, in the sense that it should not modify their workflow or ask them to learn new tools. To this end, it should happen during the first run and asynchronously with respect to the main RDataFrame computation. Through the I/O libraries in ROOT, only the columns and the *clusters* of entries that are actually processed in the computation graph will be read from the remote storage. Thus, caching systems should try to leverage this behaviour by storing a subset of the input data that is as close as possible to what RDataFrame actually reads, preferably exactly the same amount.

3.1 Caching on a file server

One aspect to keep in consideration is the physical destination of the cached files. For example, everything could be stored in a single machine, acting as caching server for the computing nodes. This approach would still require all the machines to be in the same network, preferably with a high-bandwidth connection between them, in order to have some chance to be faster than reading the files from a remote storage facility. The technology that will be tested for this configuration is XRootD. In particular, its *proxy plugin* fits in the requirements described so far. In this work, a single machine of the cluster acts as caching proxy, standing between the client application and the remote storage system. When the client asks for one or more remote files, the request will be redirected through the proxy and then to the final endpoint. Any file that is not already present on the proxy will be downloaded and stored in a specified directory.

By default, files downloaded to the proxy are prefetched in chunks. This takes some time at the beginning of the user application, slowing it down with no added benefit thus needs to be disabled. Also the size of the data chunks that are stored on the proxy can be decreased to a very low value (in this work it is set to 4 kB) in an attempt at making XRootD store exactly the equivalent in size of the TTree *clusters* of the input dataset. Furthermore, the URL of the proxy is automatically prepended to the user-supplied endpoint path, thus making the proxy completely transparent in the user application. In its final configuration, this mechanism runs as follows. During the first run, computing nodes make a request to the XRootD proxy to read a particular fragment of the input remote dataset. The proxy then fetches the requested portion remotely, caches it internally then serves it to the node which has made the request. During subsequent runs, the request of a worker node is served directly from the local cache on the proxy. This workflow is shown in Figure 2.

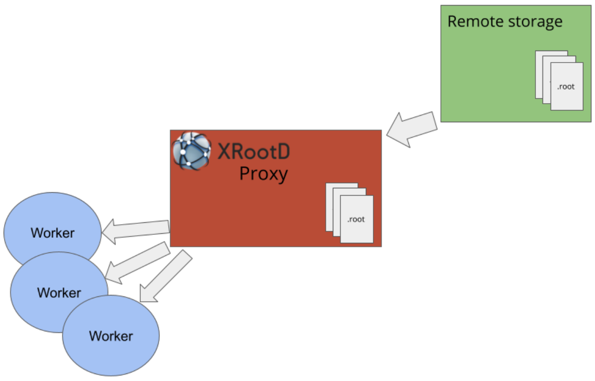

Figure 2. XRootD proxy cache. During user analysis, computing nodes (labeled "worker" in the image) make read requests for their assigned ranges of entries to the proxy server, which in turn forwards such requests to the remote storage system. The proxy stores the requested entries in its local filesystem and will be able to serve them directly to the nodes during subsequent runs of the application.

3.2 Caching on the distributed computing nodes

Making each computing node store only its portion of processed data on its local filesystem could be another approach. In order to exploit these computing-node-local caches, the

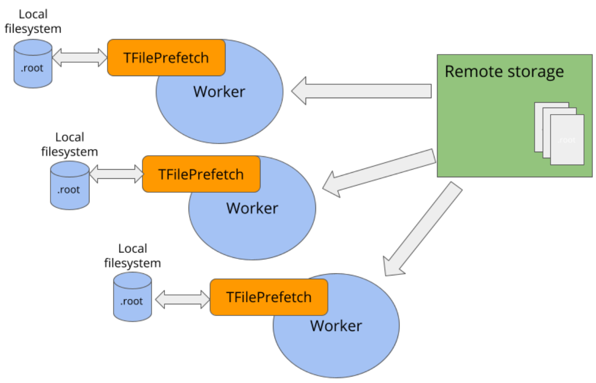

Figure 3. TFilePrefetch cache. During user analysis, computing nodes (labeled "worker" in the image) make read requests for their assigned ranges of entries directly to the remote storage system. On each node, TFilePrefetch intercepts the incoming blocks of entries and stores them on the local filesystem. In subsequent runs, each node will be able to read the same range of entries from the local disk instead of requesting it again from remote.

application scheduler needs to guarantee data locality, that means it needs to submit tasks where their input data fragments were previously cached. ROOT provides an experimental component for caching files on the local filesystem, namely TFilePrefetch. Its internal workflow is shown in Figure 3. A second thread is spawned at the beginning of the application. It is in charge of prefetching blocks of TTree entries from the remote files while the main thread is requesting them. It is completely asynchronous to the main event loop and allows to store blocks from memory to local files. This system will be activated on each computing node, caching only the necessary TTree *clusters* on the machine. The TFilePrefetch thread is only responsible for the I/O of the blocks of entries and in general does not put extra strain on the CPU running the RDataFrame computations, especially in runs where the cache is already populated. Enabling this feature involves setting two variables, namely `TFile.AsyncPrefetching` and `Cache.Directory`, either in the *.rootrc* global configuration file for ROOT or directly in the application code.

4 Test runs

The tests developed focus on showcasing the transfer of data from the remote storage system to the computing nodes or the caching server. A reference dataset has been created: a single ROOT file with a TTree of one hundred thousand entries and ten TTree *clusters* (exactly ten thousand entries per *cluster*). The dataset contains five columns of randomized data. The tests will always try to read one specific column of type `double`. The total file size is 1.8 GB, while the column of interest is 700 MB. This file is uploaded to EOS at CERN so that it will be readable through XRootD.

The RDataFrame computation graph is the same for all the tests: a very lightweight function running on the selected column of the dataset. This is enough to trigger the XRootD read requests from remote storage and observe the different effects depending on the caching mechanism enabled. The baseline is defined by running this RDataFrame application on a single machine, either with no cache or with one of the two caching mechanisms enabled. Following test configurations distribute the application to a set of nodes thanks to the Spark backend of the distributed RDataFrame interface. The source code of each test is available on a GitHub repository [14].

4.1 Hardware setup

The hardware setup is made of a physical machine plus a set of virtual machines. The baseline of tests runs on the following machines:

- 1 physical node, 4C/8T i7-6700, 256 GB SSD storage and 16 GB RAM. Serves as cache server in the tests with XRootD cache enabled.
- 1 virtual machine (VM) with 1 core, 10 GB spinning disk storage, 1024 MB RAM. Runs the RDataFrame application.

The second test configuration reuses the same physical node, but extends the number of total VMs to 5 (each with specifications identical to those of the VM described above) in order to form a Spark cluster. The virtual machines are created in the CERN OpenStack Cloud [15], while the physical machine is located at CERN. Thus, all machines used in the tests are inside the CERN network.

5 Results

In this section two different test scenarios are presented. In the first scenario, the RDataFrame application described in Section 4 is executed on the single node setup described above. In the second scenario, the same application is distributed over the Spark cluster described above. Each scenario in turn presents three tests: the baseline test with caching disabled, one test with XRootD cache enabled on a server separate from the computing nodes and one test with TFilePrefetch cache enabled on the local filesystem of the computing nodes. The results of the tests are presented in this section and are discussed in Section 6.

5.1 Single node

In the single node scenario, the reference dataset is read from EOS on the computing VM node during runtime. Only the selected column of the dataset is cached. If XRootD cache is enabled, data belonging to the selected column in the test will be stored on the caching server. The size of the cached data depends on the XRootD block size. For that reason its default value has been changed as explained in Section 3.1, so that the cache will contain approximately the same volume of data of the selected column.

If TFilePrefetch cache is enabled instead, data are stored directly in the local filesystem of the VM. This mechanism caches exactly the TTree *clusters* that the application requests. Subsequent runs will read data from the cache and not from EOS. Each test is run a thousand times to get a significant distribution of the execution times of the application, since this value might vary especially when the cache is disabled. Figure 4 shows the execution time results. For the cache enabled cases, only the runs where the cache was already populated are considered (i.e. no cold cache runs are shown).

In the same figure, it is possible to observe the incredibly higher variability in execution time of the application when reading data from EOS rather than from the caches (the standard deviation with caching disabled is respectively 13 times higher than the standard deviation with XRootD cache and 10 times higher than the standard deviation with TFilePrefetch cache). At the same time, the average execution times with cache enabled are lower, respectively by 38% with XRootD and by 48% with TFilePrefetch. See Table 1 for a summary of these results.

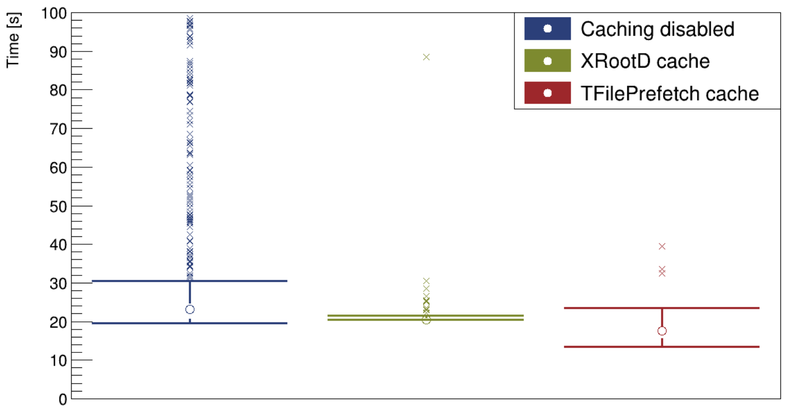

Figure 4. Single node scenario. Box plots of the distributions of execution times of one thousand test runs in three configurations: Caching disabled, XRootD cache, TFilePrefetch cache. The empty circles represent the median values of the distribution, the whiskers are drawn at 1.5 × *IQR* and the crosses outside the whisker boundaries represent distribution outliers.

The time to populate the caches was measured as well. XRootD cache takes on average 43 s with a standard deviation of 35 s, while TFilePrefetch takes on average 60 s with a standard deviation of 41 s.

Table 1. Statistics for one thousand test runs along three configurations, with the single node setup.

	mean [s]	median [s]	σ [s]
Caching disabled	33	24	28
XRootD cache	21	21	2
TFilePrefetch cache	17	18	3

5.2 Distributed cluster

In the second test configuration, all the VMs are included in the setup and it is possible to send tasks to the Spark cluster through the distributed RDataFrame interface. The Spark setup is thus made of one VM acting as Spark driver (the node from where the tests will be submitted), one acting as Spark master (the cluster coordinator and application scheduler) and the other three nodes acting as the Spark executors.

Each test is repeated 100 consecutive times, in order to simulate an interactive scenario where an exploratory user analysis is run once and then rerun subsequent times on the same data after some parameter modification. There is no need to repeat the test as many times as in the single node scenario, since it is enough to observe the caching behaviour working for a few repetitions of the application. The input dataset is transparently split in three logical partitions by the distributed RDataFrame module: it is sufficient to give an optional parameter *npartitions* to the RDataFrame constructor. Each partition is sent together with the computation graph to one of the three worker nodes as described in Section 2. Each task then reads and processes data independently of the others. When the XRootD cache is enabled, the data corresponding to the whole selected column of the dataset is stored on the caching proxy server. With TFilePrefetch enabled instead, each task caches the logical portion of the column on the node which is processing it.

The results of this configuration are shown in Figure 5. The first points of the lines corresponding to tests with caching enabled show the run during which the caching mechanisms are downloading and storing the processed portions of the dataset. In the particular case shown in the figure, the run where the cache is being populated takes roughly 10 times more than subsequent runs with both caching mechanisms. High spikes in the execution times of some of the runs with TFilePrefetch cache enabled are striking. They opened another topic of investigation in this work that will be further discussed in Section 6. This investigation led to modify the RDataFrame distributed module with the aim of forcing the Spark backend to apply data locality, i.e. to map tasks operating on the same logical range to the same node in subsequent runs. Following points of the TFilePrefetch cache line show a higher execution time than the respective points on the XRootD cache line. This was not expected but could be due to some unpredictable strain on the host machines of the VMs.

Rerunning the same tests with the improvements of forcing data locality leads to Figure 6. In this figure the cold cache runs are not shown, instead the focus is on the subsequent runs with the cache already populated. The spikes previously observed using TFilePrefetch cache are no longer present. Average execution times with the two caching mechanisms are similar and summarized in Table 2. On average, running with a cache mechanism enabled (either XRootD cache or TFilePrefetch cache) is slightly more than 2 times faster than running without cache.

Table 2. Statistics for one hundred test runs along three configurations, with the distributed setup and a locality-aware scheduler.

	mean $[s]$	median $[s]$	σ $[s]$
Caching disabled	36	26	17
XRootD cache	15	15	0.6
TFilePrefetch cache	16	16	0.5

6 Discussion

The results presented in Section 5 generally shows that enabling caching during the first run of the application makes subsequent runs faster. Each scenario and caching mechanism also show non trivial details and insights.

All test runs with caching disabled show a strikingly high variability in the execution time distribution, with a very long tail. This is a sign of the high load that storage systems like EOS have to sustain. This translates into unpredictable slowdowns in network I/O even when reading from within the CERN network as was done in this setup. This is already a strong point in favor of enabling caching for this kind of analysis, in order to protect the user from high latencies or overhead in remote data access.

In the single node scenario, XRootD cache shows the execution time distribution with the lowest standard deviation. In general this is not expected, but it is likely that the storage performance of the VM is responsible for the larger distribution in the TFilePrefetch case.

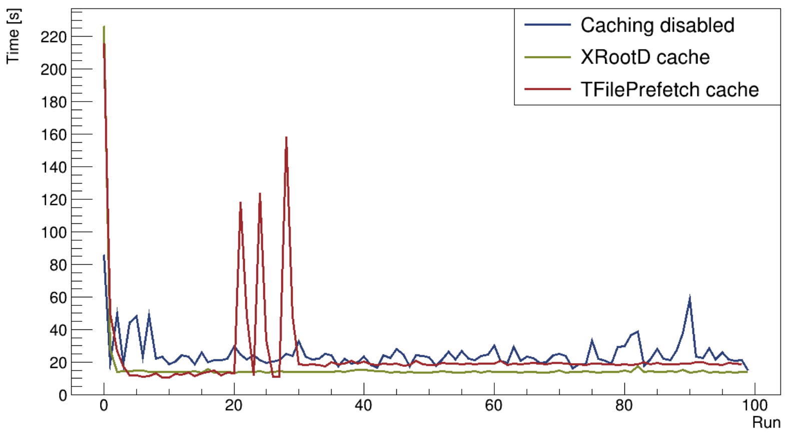

Figure 5. Distributed scenario. Lines represent the execution times along one hundred consecutive runs for three configurations: Caching disabled, XRootD cache, TFilePrefetch cache. The first point of the two configurations with cache enabled correspond to a run where the caches were being populated.

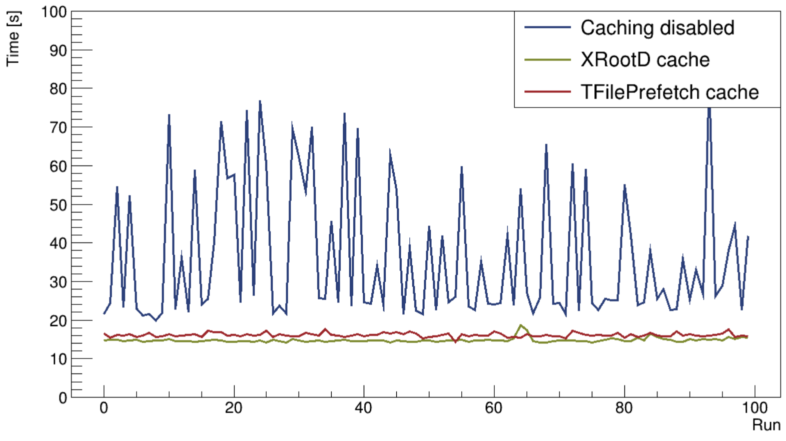

Figure 6. Distributed scenario, with data locality aware scheduler. Lines represent the execution times along one hundred consecutive runs for three configurations: Caching disabled, XRootD cache, TFilePrefetch cache. In the configurations with cache enabled, the caches were already populated in every run.

Nonetheless, TFilePrefetch shows the lowest average runtime, which is expected in general since data is stored directly on the same machine where it is processed.

The preliminary results of the distributed scenario shown in Figure 5 demonstrate that data locality is of utmost importance when caching on the computing nodes. This actually opens a new research question for this kind of effort: how to guarantee task pinning to nodes in a distributed computing environment. When distributing an RDataFrame application, the scheduler does not read the actual dataset and stream portions of it to the various nodes. Instead what the Spark master receives is a list of logical ranges of entries in the dataset and each element of that list corresponds to a task on some node of the cluster. In this context, task pinning would be beneficial. That means having the same logical range (so just a pair of integers) cached on the same worker node for all runs of the application.

Spark for example offers a `Cache` function in its API, but that still doesn't guarantee that tasks will always be sent to the same executors in the cluster. It is rather a way to signal the Spark scheduler that it is desirable to have that particular logical range cached on the cluster. In this sense, data locality is guaranteed eventually rather than at all times. It is possible though to have some stricter guarantee if some limits are set on the analysis workflow, namely that the application only runs computations of one single RDataFrame object and that the user does not exit the scope of their application until the end of their exploratory work. This was fully implemented in the distributed RDataFrame module for the purposes of this study, with no change in user code. Within this configuration, the distributed RDataFrame tests with the Spark backend indeed always pinned the same task to the same executor. This result is what Figure 6 shows, with the TFilePrefetch line overlapping the XRootD one for runs with the cache already populated.

7 Conclusions

This work integrates two caching technologies in a distributed RDataFrame application. In the distributed test scenario, both caching mechanisms give on average a factor 2 speed improvement with respect to the average baseline measurement with caching disabled.

The XRootD framework is quite well established in the community and its proxy plugin system may be used to cache remote files closer to the computing nodes. This technology is used in this work to cache input data on a server separate from the rest of the computing nodes. Another configuration makes use of the ROOT TFilePrefetch class to cache input data locally on the computing VMs that are running the application. The first configuration does not underperform with respect to the latter in the different scenarios tested. The main reason for this is the extra overhead of accessing virtualized storage on the small VMs and the difference in storage devices between the XRootD cache server (SSD) and the VMs (spinning disk). It is planned to further investigate this matter in the future with a cluster of physical machines and more homogeneous storage devices, in a setup with more geographical distance between the remote data source and the computing cluster and its caches. Another key objective for future work is to develop further on the logic for caching an input dataset during an RDataFrame analysis, possibly creating an ad hoc solution closer to the RDataFrame core.

8 Acknowledgments

This work benefited from the support of the CERN Strategic R&D Programme on Technologies for Future Experiments [16]. Particular thanks go to the WP7 coordinators, Jakob Blomer and Graeme Stewart, for their help and support during the development of this work.

References

[1] I. Bird, Annual Review of Nuclear and Particle Science **61**, 99 (2011)
[2] I.B. Alonso, O. Brüning, P. Fessia, M. Lamont, L. Rossi, L. Tavian, M. Zerlauth, Tech. Rep. 10, CERN (2020)
[3] E. Elsen, Comput Softw Big Sci **16** (2019)
[4] D. Piparo, E. Tejedor, P. Mato, L. Mascetti, J. Moscicki, M. Lamanna, Future Generation Computer Systems **78**, 1071 (2018)
[5] D. Piparo, P. Canal, E. Guiraud, X. Valls Pla, G. Ganis, G. Amadio, A. Naumann, E. Tejedor Saavedra, EPJ Web Conf. **214**, 06029 (2019)
[6] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing p. 10 (2010)
[7] V. Avati, M. Blaszkiewicz, E. Bocchi, L. Canali, D. Castro, J. Cervantes, L. Grzanka, E. Guiraud, J. Kaspar, P. Kothuri et al., Euro-Par 2019: Parallel Processing pp. 241–255 (2019)
[8] A. Peters, E. Sindrilaru, G. Adde, Journal of Physics: Conference Series **664** (2015)
[9] A. Dorigo, P. Elmer, F. Furano, A. Hanushevsky, WSEAS Transactions on Computers **4**, 348 (2005)
[10] L.A.T. Bauerdick, K. Bloom, B. Bockelman, D.C. Bradley, S. Dasu, J.M. Dost, I. Sfiligoi, A. Tadel, M. Tadel, F. Wuerthwein et al., Journal of Physics: Conference Series **513** (2014)
[11] A. Hanushevsky, H. Ito, M. Lassnig, R. Popescu, A. De Silva, M. Simon, R. Gardner, V. Garonne, J. De Stefano, I. Vukotic et al., EPJ Web Conf. **214**, 04008 (2019)
[12] *ROOT TFilePrefetch class reference guide*, https://root.cern/doc/master/classTFilePrefetch.html (2011), accessed: 2021-06-03
[13] V.E. Padulano, J. Cervantes Villanueva, E. Guiraud, E. Tejedor Saavedra, EPJ Web Conf. **245**, 03009 (2020)
[14] *rdf-dist-cache GitHub repository*, https://github.com/vepadulano/rdf-dist-cache (2020), accessed: 2021-06-03
[15] *CERN OpenStack overview*, https://clouddocs.web.cern.ch/overview/overview.html (2021), accessed: 2021-06-03
[16] M. Aleksa, J. Blomer, B. Cure, M. Campbell, C. D'Ambrosio, D. Dannheim, M. Doser, F. Faccio, P. Farthouat, C. Gargiulo et al., Tech. rep., CERN, Geneva (2018)