EPJ Web of Conferences **177**, 05003 (2018) https://doi.org/10.1051/epjconf/201817705003
AYSS-2017

DataForge: Modular platform for data storage and analysis

Alexander Nozik*

[1] *Institute for Nuclear Research RAS, prospekt 60-letiya Oktyabrya 7a, Moscow 117312*
[2] *Moscow Institute of Physics and Technology, 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russian Federation*

Abstract. DataForge is a framework for automated data acquisition, storage and analysis based on modern achievements of applied programming. The aim of the DataForge is to automate some standard tasks like parallel data processing, logging, output sorting and distributed computing. Also the framework extensively uses declarative programming principles via meta-data concept which allows a certain degree of meta-programming and improves results reproducibility.

1 Introduction

Modern experimental research, both in the field of particle physics, and in other fundamental and applied problems, is usually associated with the acquisition, storage and analysis of large amounts of data. In all three stages of working with data (obtaining, storing and processing) in modern conditions, the maximum level of automation is required in order to both save the working time of highly qualified employees, and reduce errors caused by manual work with data. To provide automation, in turn, it is necessary to have specialized software that would allow, on the one hand, to solve the entire set of tasks for working with data, on the other - it would not require high programmer skills from users.At present, there are a few data analysis platforms, but most of them were created in the early 2000s or even earlier, and as a consequence, they do not use all the variety of software tools that have emerged since then. In addition, almost all existing tools were created for a highly specialized task and as a result are poorly suited to work with the entire chain: acquisition, storage and analysis.

In particle physics the most used framework for data storage and analysis is ROOT ([1]). The system is developed and supported by CERN and is used in most of modern large-scale experiments. The framework is monolithic, implements obsolete architecture and also has a lot of minor and major flaws, discussed in the literature. There were a few attempts to create a better alternative for ROOT like JAS3([2]) or DataMelt, but none managed to solve all of the problems. Software solutions for data acquisition are usually commercial projects with closed code, which makes it difficult to create a more or less universal system. The DataForge is a project intended to address these problems, namely provide modern lightweight framework to perform the full cycle of data processing without a lot of programming expertise.

*e-mail: nozik@inr.ru

2 Basic concepts

DataForge ([3]) introduces a several major principles which serve both as ideological guide for developers and design hints for users.

The main one is the principle of **metadata processor**. According to this principle, during the processing of data, only data (immutable initial data in any textual or binary format) and metadata (user-defined tree of values) are allowed to be used as input, meaning no user instructions (scripts) or manually managed intermediate states are possible. This means that user is forced to use declarative approach to describe what should be done instead of imperative one. Of course such restriction limits user capabilities, but on the other hand, processing description in the form of metadata could be a subject of programming itself. One can automatically transform metadata, use it to cache analysis results or even send instructions to remote computing node. All these features are hard to implement using scripting approach. Automatic script transformation is unreliable and requires sophisticated code processing tools like parsers and syntax analyzers. While remote procedure call is possible via scripts, in most cases it is restricted to simple imperative commands. Here we give only three examples of the usefulness of the metadata processor:

1. The results of the analysis are completely determined by metadata. This means that each result is mapped to a fixed configuration. Configurations, unlike scripts, can be compared. If the configuration of some part of the analysis has not changed, it can be considered guaranteed that the result of this part has not changed. This result can be calculated once, placed in the cache and then used ready, significantly saving computational time.

2. It is frequently necessary to do some analysis for a list of different values of some parameter. In most cases, this is done with the help of macros recording a sequence of user actions. Macros are generally cumbersome and not reliable. In the case of a metadata processor, it is sufficient to make a metadata generator that will change one or more values in the source tree and run an analysis for all these configurations.

3. Performing distributed computing, one of the biggest challenges is ensuring the identity of the execution environment and the set of instructions on all nodes of the network. Metadata automatically solve the problem of instruction identity (it is just a tree of values that can be easily compared). For the identity of the environment, the principle of context encapsulation is used (see below).

Separately, it is worth noting the mechanism of layering: two or more metadata sets could be automatically combined without changing individual configurations. Metadata of a higher level overlap certain values of the metadata of a lower level, without changing the lower level itself. The layering system is also very convenient for selective configuration of individual data sets. For example, if there is a general configuration for the entire process, but additional instructions are required for a specific element, rather than overwriting the entire configuration and processing this element separately, a small layering is created that affects only this element. At the same time, all analysis is done, as before, and can be processed in parallel if it is possible.

The second principle is **context encapsulation**. One of the main difficulties in the development of parallel and distributed computing is the influence of the environment (connected libraries, global variables, object states) on the results. DataForge solves this problem by introducing the concept of context. The context contains all I / O tools and connected modules. The context is organized in such a way that the identity of the two contexts can be easily verified (the context description can be transformed into metadata). Neither the context, nor any connected modules can be changed during

operation. Thus, processes can not affect the results of other processes. Also, any acquisition, storage or analysis parameter could be accessed from the context using its name. A context encapsulation is extensively used in Android SDK.

Finally, a very important feature of DataForge is its **modularity**. The core of the system contains only basic functionality for working with metadata and organizing the data processing, all specific functions, such as the graphical environment or tools for statistical analysis, are placed in separate plug-ins. It is planned that modules can be dynamically loaded from a remote repository as needed (such a system is implemented, for example, in JAS3 - [2]). Modularity is the standard for software in Computer science, but monolithic systems are still very often used in experimental physics. Monolithic systems are mostly very complex in terms of support and upgrades.

3 Data flow

Metadata processor approach requires a lot of attention to be paid to data flow models. DataForge currently considers two of them:

- **Push-model**. This model actually implements the classical concept of map-reduce. Processing consists of a sequence of actions on the data tree, in each of which a certain transformation of the data takes place. If individual tree elements can be handled independently, then the process automatically (without additional actions from the user) is performed in parallel. The fundamental innovation of DataForge in this data flow model is the process of configuring the actions. The user specifies some configuration of each of the actions, but the metadata of the elements of the data tree themselves overlap the configuration of the action. Thus, if the metadata of an element contains information related to the action, then this information will be automatically taken into account without writing a special processing instruction. This feature significantly simplifies the analysis of large amounts of data when additional external information is required (for example, calibration in physics experiments).
- **Pull-model**. This model is fundamentally new in data processing. Its essence lies in the fact that the user provides some configuration of the result that he wants to see, and the system itself determines what actions and what data it must request to obtain the result. The relations between processing blocks in pull model are called dependencies. When some pull block is called, it first calculates a set of its dependencies based on its metadata and then requests appropriate data or pulls other blocks, using input metadata to produce configuration for dependencies.

Both models involve the use of lazy computing (where possible). The usual procedure is to use pull model globally to define global data processing blocks - tasks and their dependencies and then use push model inside single task. Usually all results of intermediate operations are automatically cached, meaning that calling task or push-action with the same set of metadata does not compute and instead loads results from cache. Since the only thing needed to perform computation is metadata, one can easily delegate computation itself to a remote node or a whole distributed computation network.

Pull data flow model is not so well known as push model and requires additional explanation. An example of existing implementation of the idea is the gradle build system ([4]) used for a wide variety of JVM-based and native projects. In DataForge it is implemented by so-called workspaces and tasks. The workspace is a virtual structure that contains an initial data tree as well as a list of tasks that could be performed. The task execution is performed in three steps:

1. **Task model**. Depending on task definition and input metadata, the system calculates an acyclic dependency graph of tasks and data. Task dependency is represented by the name of task and metadata passed to the task (task could modify metadata passed to dependencies), so each task

calculates its dependencies recursively. Data dependencies consist only of data name or mask. Task and data are resolved by the workspace by name. Task graph is checked for cycles.

2. **Lazy computation model**. A 'Goal' object is created for each specific computation. Goal is basically an elementary lazy computation result which depends on completion of other goals. A task could produce one or many goals for each of data pieces passing through it. There could be pipe goals that transform one input element into one output element, join goals, split goals as well as custom goals with more sophisticated dependency mechanism. Goals are organized in acyclic graph (the complexity of this graph is generally much larger then the one for tasks and user does not have control over it beyond basic canceling). In theory, goal graph could be automatically optimized, but for now there was no practical reason to do it.

3. **Computation**. When the top level goal is triggered, it invokes computation of all goals in chain behind it. If some of goals in cycle is not triggered, it won't be computed thus saving computation time without need for the user to explicitly optimize the data flow. If for some reason, one of the goals is already computed (or just started) when it is triggered, it is not computed again, the previous result is returned. If caching mechanism is enabled and there is a cached goal result with the same medatada, it is restored from cache instead of computation. It is possible to have not only local, but also remote cache, which allows to implement sophisticated distributed network systems.

4 Current state and future plans

Current prototype of the DataForge is written on JVM platform and uses Java 8 and Kotlin languages for core modules. The JVM automatically grants a wide cross-platform portability and by default supports major operating systems like MS Windows, Linux and MacOS. Also JVM is known for its good tooling support, which simplifies development, and has good internal multi-threading support, which is needed for parallel computation. Performance issues experienced in earlier versions of JVM platform were almost completely eliminated in last years and currently JVM just-in-time compiled code has almost the same performance as native C++ code (the statement is bases on multiple case-study articles since direct comparison is not available due to differences between JIT and compiled code operation).

Since the most of DataForge functionality could be accessed via manipulation with metadata trees, it is quite easy to create a client application in any other language which can transform request into metadata, send it to DataForge server and then transform result back into user-friendly representation. It is planned to create thin Python-based client as well as web-based console in the future. Current prototype also supports basic communications with native processes on local computer as well as TCP/IP communication with remote computers or devices and serial port communication (a universal API for most available hardware and software architectures).

Since developing a stand-alone framework for both data acquisition and analysis is a very large scale task, current aim of DataForge is not to completely replace existing tools, but to build additional configuration layer on top of them. Here are examples of such uses for different subsystems:

- **Parallel computations**. Goal API allows to use different implementations for task scheduling and running. DataForge currently utilizes Java 8 fine threading based on fork-join thread pool ([5]) as a default. It also has another implementation using experimental coroutine framework ([6]) developed by JetBrains. It is planned to also implement a back-end for Apache Spark framework ([7]).

- **Caching**. Caching is important part of DataForge since it allows to avoid recalculation of already completed tasks. The caching system utilizes JCACHE specifications ([8]) and thus allows to use different powerful caching packages (though by default it uses internal lightweight implementation).
- **Storage**. The storage API of DataForge is abstract and allows to use different local and remote storage mechanisms like SQL and no-SQL databases. The important distinction is DataForge treats all data as immutable that allows to significantly simplify interaction with storage software.
- **Control**. Device control API uses abstract definitions based on the concept of virtual device and its states. It is planned to introduce a connection to the TANGO slow control system ([9]) as a back-end for large scale projects.
- **Visualization**. DataForge has a small internal tool set for creating graphic user interfaces, but in future it will be mostly replaced by web-based visualization using existing libraries.

Before 2017 DataForge was developed as a part of software package for "Troitsk nu-mass" experiment in search for masses of active and sterile neutrinos ([10]). The experiment data requires a very complicated and multi-step analysis and DataForge allows to greatly simplify the process and avoid making mistakes that would be hard to find. Currently, DataForge is used for the data acquisition, storage and control in this experiment. Currently it is being adopted for simulations and analysis management in center for nuclear physics methods at MIPT ([11]).

5 Conclusion

The DataForge framework is still in early stage of its development, but it demonstrates new approaches to scientific software development (both from ideological and architectural point of view). A metadata processor ideology imposes some limitation on user and developer actions inside the framework, but opens a lot more possibilities like meta-programming, automatic result caching, distributed computing etc.

References

[1] *ROOT homepage*, https://root.cern.ch/ (2017), [Online; accessed 24-January-2018]
[2] M. Donszelmann, T. Johnson, V.V. Serbo, M. Turri, Journal of Physics: Conference Series **119**, 032016 (2008)
[3] *DataForge homepage*, http://npm.mipt.ru/dataforge/ (2017), [Online; accessed 06-November-2017]
[4] *Gradle homepage*, https://gradle.org/ (2017), [Online; accessed 06-November-2017]
[5] *Package java.util.concurrent*, https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/package-summary.html (2017), [Online; accessed 24-January-2018]
[6] *Concurrent Programming in Kotlin: Coroutines*, https://dzone.com/articles/approaching-kotlin-coroutines-concurrent-programmi (2017), [Online; accessed 24-January-2018]
[7] *Apache Spark homepage*, https://spark.apache.org/ (2017), [Online; accessed 24-January-2018]
[8] *JCACHE: JSR-107*, https://www.jcp.org/en/jsr/detail?id=107 (2017), [Online; accessed 24-January-2018]
[9] *TANGO controls*, http://www.tango-controls.org/ (2017), [Online; accessed 24-January-2018]
[10] D.N. Abdurashitov et al., JINST **10**, T10005 (2015), 1504.00544
[11] *Center for nuclear physics methods at MIPT*, http://npm.mipt.ru/ (2017), [Online; accessed 24-January-2018]