Proceedings of the XXVI International Symposium on Nuclear Electronics & Computing (NEC'2017)
Becici, Budva, Montenegro, September 25 - 29, 2017

DATA MANAGEMENT IN HETEROGENEOUS METADATA STORAGE AND ACCESS INFRASTRUCTURES

M.V. Golosova[a], E.A. Ryabinkin

National Research Center "Kurchatov Institute", 1, Akademika Kurchatova pl., Moscow, 123182, Russia

E-mail: [a] golosova.marina@cern.ch

The paper is dedicated to the problem of data management in complex infrastructures for data storage and access. This kind of infrastructures commonly appears in long-term projects and is designed to bring order and simplify information lookup in the (semi-)independent parts of the original information structure. In this work one of the basic tasks of such structures is discussed: the organization of the integration of different types of information, available in the information field of the project, within a number of special integrated storages. Authors provide a conceptual view and first prototype of the system being developed in order to make this process well-organized and easily tunable for any set of the original information sources and integrated storages, common and project-specific data transformation tasks.

Keywords: metadata integration, knowledge base, Apache Kafka

1. Data Integration and metadata processing

Data integration (DI) is one of the most commonly appearing and important tasks in the field of IT since the beginning of the massive utilization of computing technologies for business or scientific projects. It provides the possibility to perform analytics basing on data gathered from different information subsystems (databases, spreadsheet, documents, text files, public repositories, etc) and to create aggregated representations, such as reports, summary or high-level overview of some processes [1].

DI is required for different types of projects, appearing in numerous fields – from small business to international corporations and scientific experiments. In the recent years a lot of powerful tools for data integration have appeared – such as Pentaho Kettle [2], Talend [3] and Informatica Data Integration Hub [4]. They provide solutions for data integration from and into wide variety of standard data sources. However there are some limitations, as well: for example, for data transformation during the ETL[1] process, even though it is possible to write custom procedure, developers are limited to a set of supported languages (e.g., JavaScript and Java for Kettle).

In ATLAS [5] R&D project Data Knowledge Base (DKB) [6], aimed to join and process data from multiple subsystems of ATLAS information infrastructure and provide a fast and flexible access to them, the ETL modules were developed independently from each other, addressing different directions of the DKB conceptual architecture development. They had to be developed as fast as possible to explore and prove various concepts, so for every task developers chose the most appropriate and most familiar programming language – Shell, Python, PHP, JavaScript, etc. Developed modules shared only the input/output data format and at first were joined via files with intermediate data; later – with pipes.

Soon the organization of reprocessing of test data sets (after introducing of new features or bug fixes in the ETL process) started to become very time-consuming. In addition, some ETL scenarios needed to be run on a regular basis to keep inner DKB storages up to date, synchronized with the original storages. Instead of addressing this for every scenario individually, it was decided to delegate the tasks to some management utility, which would orchestrate all the ETL processes and make data (re)processing and update in an automated way. The main problem was to provide reliable data transfer between stages, make the process more seamless – yet stoppable and restartable at any moment without a need to reprocess a lot of already processed data. Also, there was a requirement to minimize changes in already existing and tested modules during their port to the new framework (and therefore, avoiding full rewrite for most of them just to obey restrictions imposed by ETL engines, especially in the area of possible programming languages).

Data that pass through the ETL processes within DKB are actually the metadata describing ATLAS project itself, scientific results of the collaboration and the inner processes of physical data production and analysis. Metadata are not that different from any other data, but there are some specifics relevant to the subject at issue:

- there are a lot of connections between metadata units, retrieved from different sources, because they may describe same objects or processes from different points of view[2];
- however described objects and processes of the same type are loosely linked to each other, meaning that metadata units from the same source in most cases can be considered completely independent from each other.

It makes ETL processes very branchy, but also allows to process metadata unit by unit, with possibility to stop at any moment and restore the processing later almost from the same point.

All the above leads to the conceptual view of the ETL management system that will be described in details in the next section.

---

[1] Extract, Transform, Load

[2] Certain data may originate from more than one system, so there is a room for collisions. Possible solutions for this problem is also a very interesting theme, though lying beyond the scope of the current work

2. ETL management system

As there are different use-cases within the DKB, connected to different metadata sources and with different access scenarios for the integrated data, there are different chains (and more complicated topologies) of ETL modules. One of the ideas of the ETL management system (MS) was to avoid redoing essentially the same machinery, but with its own management rules, configurations and start/stop scenarios for each use-case. It would complicate management of the project as a whole and in the end lead to another set of independent information systems – exactly the same situation that DKB is supposed to overcome.

The task for ETL MS could be formulated as follows:

- to join independent ETL modules, developed specifically for certain purposes, into a fully automated ETL system providing mechanism for initial data load as well as the synchronization mechanism, that would keep integrated storages within DKB up to date with the original metadata sources;
- to provide a possibility of adding, removing and replacing of individual modules in a more flexible way than updating the source code of the supervising utilities;
- to provide reliability and scalability for all the ETL processes within DKB:
    - reliable data delivery between ETL modules:
        - exactly once;
        - fault-tolerant;
    - basic parallelization (by metadata units), configured via the MS, not within the modules themselves;
- to simplify the development of new ETL modules (and updating of the already-existing ones) as much as possible by:
    - leaving only the project-specific tasks for implementation;
    - providing a set of requirements for the modules intended to be inserted into the ETL processes, made as simple to meet as possible.

2.1. Stream processing

The first fundamental step was to replace bulk processing with the streaming approach. There were three main reasons for this decision:

- the synchronization mechanism requires a possibility to process not *all* the data from the initial sources, but only the new and updated data; it means that one cannot count on the possibility to process all the data at once (e.g. for aggregation operations);
- the volumes of data stored in ATLAS IS[3]es make it unwise to try processing *all* the data step by step: it would unreasonably raise the requirements to the temporary storages between processing steps and wouldn`t allow to see any result before all the steps finished completely or ensure that all the nodes in the ETL topology is working and configured properly;
- to provide fault tolerance for the whole process and to avoid massive reprocessing it still would be necessary to split *all* the data into a smaller chunks.

2.2. Pluggable modules

As soon as the stream processing became the basic concept, a whole world of stream processing technologies was there to investigate. Due to the requirement to keep set of ETL modules multilingual, the next landmark decision was to leave ETL modules to exist as independent programs, but with one provision: they must be pluggable into the streaming system. This way the development process changed very little, allowing developers to choose the way to address each task individually, as they used to. And the most notable change affected the requirements to the way these modules consume input data and output the results.

The requirements for the pluggable module (italics describe rationale for each) were:

[3] Information System

- input and output go through the STDIN/STDOUT; log messages – to STDERR *(as if they were intended to interact in the pipeline);*
- waiting for input messages in the infinite loop *(so that it can process as many messages as needed and as soon as they are available, without destroying and recreating the process each time new data arrived);*
- processing one input message at a time *(so that only one message is to be reprocessed at restart after failure);*
- support common flow control specification *(to avoid message congestion in the STDIN/STDOUT pipes).*

These requirements are to be refined after investigating the potential side effects. Some of them are already known or can be easily predicted, such as:

- even when there are no data to process, all the ETL subprograms will still be running, in the standby mode. If it affects the system in some way, the processes can be destroyed when there are no new data to operate with – but this will become a responsibility of the MS, not pluggable modules;
- the restriction on the number of messages being processed simultaneously within the topology of ETL modules leads to the restrictions to the whole processing speed: if there is one slow node in one branch of processing topology, all the other nodes will have to wait for it and the whole topology will operate with the slowest node pace. It can be solved by accurate topology design, isolating slow branches into separate threads or processes.

2.3. ETL module specifics

On the example of the ETL modules developed for DKB project, it was understood that there are three types of modules, differing in the basic functionality. There are separate modules for data *extraction*, for *processing* (or transformation) and *load*. The "extraction" modules were to be executed on a regular basis, while "processing" and "load" ones were to operate as soon as new data are available. For the "load"-type modules it is vital to work in a batch mode, if the integrated storage into which the data are to be loaded supports bulk upload – otherwise the loading process can become very slow.

3. Prototype

3.1. Underlying technology

In the quest for the basic technology, the main criteria were as follows:

- the technology must provide possibility of joining any number of modules into some kind of a topology, where data flow runs from *sources* (extraction modules) to *sinks* (loaders) through a number of *processors*;
- the topology must provide the possibility to reuse parts of the data flow to avoid reduplication (say, streams of data after processors P1 and P2 must be processed by program P before going to sink S; instead of creating two instances of processor P for every stream, these two streams could go through the same processor);
- the technology must provide a reliable mechanism of data delivery between *nodes* of the topology;
- the technology must support the horizontal scaling:
    - work in distributed mode, allowing running different parts of the ETL topology (*subtopologies*) on different servers;
    - scale up for more intensive input data streams by adding more servers to the cluster.

The technology, chosen for the prototype, was *Apache Kafka*. Apache Kafka is a free software, combining features of message delivery (data streaming) and stream processing systems, which is exactly what was needed to fit the requirements.

As a data streaming platform, Apache Kafka provides concepts of source and sink *connectors* (matching the ETL "extract" and "load" modules) and *topics* – temporary storages for message

queues. Connectors interact with topics via *producers* (ones to publish new messages into topics) and *consumers* (ones to subscribe to the topics and read new messages as soon as they are available).

As a stream processing platform, it provides a special library named *Kafka Streams* that implements the basic functionality of *processors* (matching ETL "processing" modules) and *topology*, the way to describe the order in which the processors are to be applied to a data flow. It also provides *state stores*, allowing *stateful* processors, necessary for aggregation operations (such as data filtering). The processing topology is linked to connectors via topics (see section 3.3 for an example and a picture with all mentioned elements).

3.2. Apache Kafka extensions

An additional functionality, needed for the configurable ETL system with pluggable modules, was developed on top of the basic Apache Kafka API: adapters (to plug ETL modules) and Streams topology supervisor with flexible topology configuration.

The adapters (External Source and Sink Connectors, External Processor) are running as wrappers for the processing modules, taking care of the interaction with Kafka objects (writing to and reading data from topics, getting input data and pushing the results of processing further along the data flow topology), running the external program at startup (or by schedule, in case of Source Connector) and restarting it on failure according to the *retry policy*. The external programs are used for the project-specific operations, running in the background and awaiting new data to arrive or producing data by request.

The Streams supervisor (a program, utilizing specially developed Configurable Topology instead of the static workflow, which was to be hardcoded for each occasion individually) takes care of creating the processor topology according to the configuration file and starting its nodes. As the nodes are started, all the external programs are run in the background, and wrappers start to push data through the topology.

With these components, to create a new ETL pipeline one needs to describe it with configuration files, telling Kafka adapters which programs (independently developed ETL modules) to run as the external processes and how to interlink them. After that, to run the pipeline one just launches the core ETL Java applications for connectors and processing topology.

3.3 Prototype

The ETL process implemented in the prototype is the one of DKB for document metadata processing (Fig. 1). There is one source connector, three transformation chains and one sink connector – 5 independent processes, linked to each other via Kafka topics. All of them during the testing were run on a single computing node[4], but technically it might be 5 different nodes as well.

First, all transformation chains were joined into a single topology, without intermediate Topic-2. But as there is one very time-consuming process (Stage 3.0), all transformation processes used to operate with its speed. After splitting the topology into three separate chains, the situation has changed: the results of Stage 5.5 were available just a few seconds after the data are published to the Topic-2, while the parallel chain kept on working for days. To speed things up, this chain was configured to run in the parallel mode.

Every of these 5 processes required about 0.5 GB of memory, so the parallelization was done via threads within the single process – but it can also be done by running multiple copies of the same process, on different nodes or on the same machine: Kafka internals will uniformly distribute the data between these processes.

[4] 48 GB RAM, 2 CPU (4 cores with hyper-threading, 2.4 GHz)

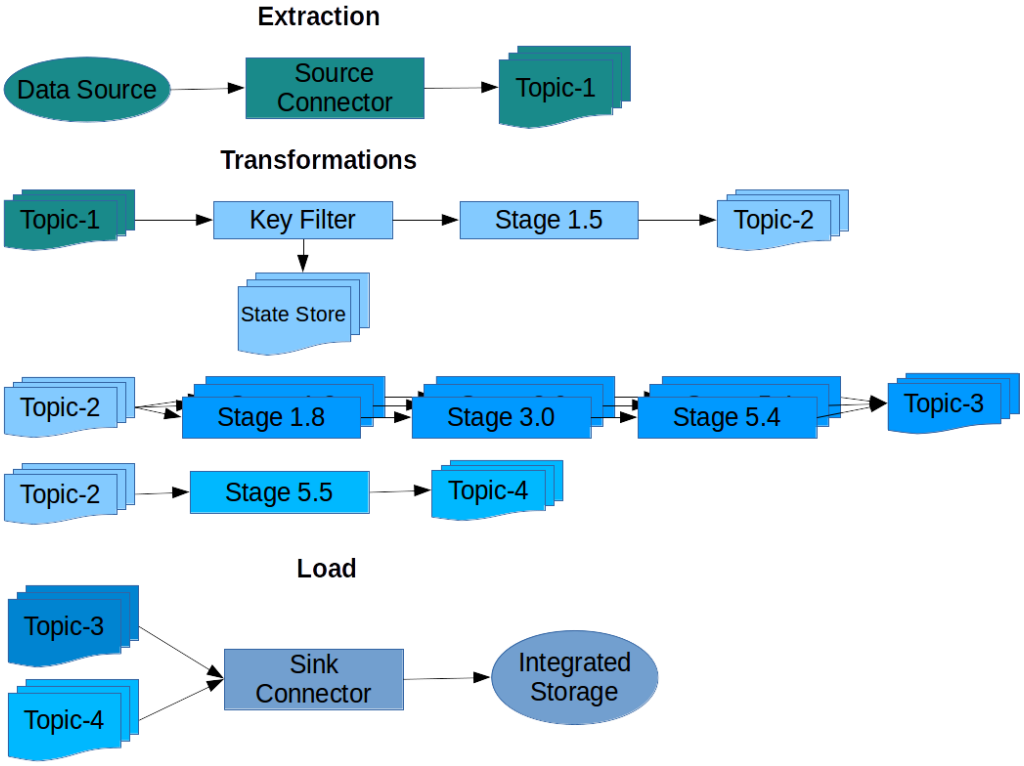

Figure 1. DKB document metadata ETL processes

The original data source provides information only in a bulk mode, without possibility to get only new or updated records. To avoid processing same data multiple times, the whole processing starts with filter processor (Key Filter in the first chain), which has its own State Store, provided by Kafka, that allows to save the process state on shutdown. It guarantees that the filter will work correctly, wherever and whenever the process is restarted. The state store volume for ~900 records is about 100 KB, while after Stage 1.5 the volume of the stored in Topic-2 data increases significantly, taking ~300 MB on the disk. If there were no filtering, this volume would grow very fast with every update from the source connector.

Data written to the Topic-1 are not filtered, so its volume inevitably grows. But it is not as noticeable as for Topic-2: for 17 thousand records it takes about 7 MB. To limit the size of the topic, it can be configured so that records that are older than a given interval (a day, or an interval calculated as two poll intervals of the source connector, or any other reasonable value) are dropped. It is safe to drop data after they are processed by the first chain and the results are written to Topic-2, as the original data, stored in Topic-1, are no longer needed (except for the reprocessing during the tests). In this case the process of initial load took about 2 hours, and then less than a minute for succeeding updates.

Of course, there is a time penalty for using Kafka adapters with external processes instead of chaining them via pipes – or instead of writing all the procedures in Kafka native language, Java.

The initialization of every process takes ~2 sec; communication between stages and with topics (during the processing of a single input record) takes ~1 ms for the shortest chain, operating in a single-thread mode. For the longest chain, running in 4 or 2 threads, this value was very much the same, but there were some spikes with values up to few hours. However when the chain was running in 1 thread, the peaks disappeared, so they are believed to arise due to a server resource bottleneck. Much more significant overhead is related to the usage of Kafka adapters and communication between them and external processes. Indirect measurements showed that it takes about 0.7 seconds per input record for every processing node of the topology. It does not look bad for the slow nodes, when processing itself can take hours and there are not so many records in the stream after every update

from the source – but for fast processing it can become much more meaningful, so it is to be studied during the follow-up works.

3.4. Adaptation of DKB modules for ETL MS

One of the tasks of the ETL MS is to simplify the transition of existing software to its infrastructure and to allow for convenient development of (at least) ETL-related parts for the new software. Existing experience in the field is gathered in this section.

As most of the ETL modules for DKB are written in Python, the requirements listed in the section 2.2 were implemented in a common library (pyDKB) (~800 NCSLOC, including some optional, but useful machinery like processing of local or stored in HDFS files and operating in the appropriate for Hadoop map/reduce jobs mode). In the most complicated cases, these requirements also initiated modifications of the input and output data format – as some valuable information used to be passed through filenames or personally between developers, and now it had to be moved into the input/output data. For these cases it took about 100 NCSLOC to make original code to operate as a pluggable module, while the rest of the original code (up to ~1700 NCSLOC) was left untouched.

For other modules, initially developed in a more stream-oriented way, changes were mainly connected to the delegation of the input/output handling to the common modules.

Some new modules were also developed in this paradigm by a person, who was unaware of the whole story but managed to complete the task by using pyDKB without visible problems.

4. Future plans

Despite fitting basic needs of the DKB project already, the ETL system has numerous directions for the further development.

1. Additional studies and improvements to reduce run-time overhead of Kafka wrappers.
2. Data transfer protocol, developed for interaction between adapters and external ETL modules, has some limitations and weak points; ways of its improvement are under discussion. It has to be kept as simple as possible, not becoming a noticeable handicap for writing new ETL modules without usage of special libraries pre-developed for every given language – yet its modifications may lead to very useful possibilities for the ETL system as whole.
3. Standard processors, implementing data flow filtering, joining, splitting, etc. They would make the topology configuration even more flexible and provide project-independent implementation for basic operations.
4. User interfaces for topology overview and configuration would be very helpful in terms of usage of the development results in projects other than DKB. Basic concepts were made project-independent, meaning that the ETL system is not limited to the original project it was first designed for.
5. Low-impact measurement framework both for ETL/Kafka infrastructure and adapters/modules to allow quantitative analysis of processing flow and "hot points" identification/elimination.
6. Monitoring interface to visualize the data flow through the ETL system, with possibility to detect errors or bottlenecks in the inner processes.
7. Automation of Kafka/ETL deployment using existing best-of-the-breed tools; Puppet-based approach is considered to be appropriate.
8. More infrastructure-oriented tests and determination of server hardware/software requirements and best practices for ETL.
9. Strategic review of the current implementation and its possible generalizations to allow for the applicability to the broader range of projects (including field testing/tuning of the said next-generation stack (items 1-8) in the context of more real-world projects of a diverse nature and formulating best practices for different use-cases).

5. Acknowledgement

The work was supported by the Russian Ministry of Science and Education under contract № 14.Z50.31.0024.

This work has been carried out using computing resources of the federal collective usage center Complex for Simulation and Data Processing for Mega-science Facilities at NRC "Kurchatov Institute", http://ckp.nrcki.ru/.

6. Conclusion

ETL methodology from data integration field was found to be useful in the context of DKB R&D project, since it allows to separate ETL infrastructure and code from the business logics and processing machinery. ETL management layer was built on top of the Apache Kafka; it allows one to provide configurable processing topology, simplify usage of Kafka features and keep existing languages for processing modules. Next, this layer was generalized to allow for its usage in the context of other projects, since it bears much resemblance to the pipeline concept (e.g. found in Unix environments); was formulated the set of requirements, built the architecture based on them and provided the actual implementation. The latter was studied in the framework of DKB code: both adherence to the formulated requirements and operational characteristics.

Basing on the obtained results, we believe that the current state of the produced ETL MS is general enough to find its place in other projects, so we had outlined further directions in its development and are planning both for a

- strategic work on enhancing the basic concepts and finding more real-world applications benefitting from this approach;
- tactical work on adding/enhancing various aspects of the system which were (and will be) found during architectural and applied development phases.

One specific piece of future work we want to explicitly mention is the more thorough search and review of Kafka alternatives for the low-level substrate. Such an approach fits our view on the simplicity of business logics integration and language-agility of ETL: it is just applied to the other side of the system.

References

[1] Casters, M.; Bouman, R..; van Dongen, J.; Building Open Source ETL Solutions with Pentaho Data Integration, Wiley, 2010, ISBN 978-0-470-63517-9
[2] http://www.pentaho.com/product/data-integration
[3] http://www.talend.com/products/talend-open-studio
[4] https://www.informatica.com/products/data-integration/integration-hub.html
[5] ATLAS collaboration, ATLAS: letter of intent for a general-purpose pp experiment at the large hadron collider at CERN, CERN, 1992, Letter of Intent CERN-LHCC-92-04, CERN-LHCC-I-2
[6] M A Grigorieva et al. Evaluating non-relational storage technology for HEP metadata and meta-data catalog // 2016 J. Phys.: Conf. Ser. 762 012017