EPJ Web of Conferences **295**, 04029 (2024) https://doi.org/10.1051/epjconf/202429504029
*CHEP 2023*

# Job splitting on the ALICE grid, introducing the new job optimizer for the ALICE grid middleware.

*Haakon André* Reme-Ness[1,*], *Costin* Grigoraş[2], *Håvard* Helstrup[1], *Bjarte* Kileng[1], *Maksim* Storetvedt[2], and *Latchezar* Betev[2]

[1]Faculty of Engineering and Science, Western Norway University of Applied Sciences, Bergen, Norway
[2]CERN, Geneva, Switzerland

**Abstract.** This contribution introduces the job optimizer service for the next-generation ALICE Grid middleware, JAliEn (Java Alice Environment). It is a continuous service running on central machines and is essentially responsible for splitting jobs into subjobs, to then be distributed and executed on the ALICE grid. There are several ways of creating subjobs based on various strategies relevant to the aim of any particular grid job. Therefore a user has to explicitly declare that a job is to be split, and also define the strategy to be used. The new job optimizer service aims to retain the old ALICE grid middleware functionalities from the user's point of view while increasing the performance and throughput. One aspect of increasing performance is looking at how the job optimizer interacts with the job queue database. A different way of describing subjobs in the database is presented, to minimize resource usage. There is also a focus on limiting communications with the database, as this is already a congested area. Furthermore, a new solution to splitting based on the locality of job input data will be presented, aiming to split into subjobs more efficiently, therefore making better use of resources on the grid to further increase throughput. Added options for the user regarding splitting by locality, such as setting a minimum limit for a subjob size, will also be explored.

## 1 Introduction

During Run 3 the ALICE collaboration[1] produces a larger amount of data, following hardware upgrades[1] and an increase in collision rate following higher luminosity[2] compared with the conditions pertained in Run 2. As a result of this, and predicting the trend will continue to show an increase in the data produced in the future, a decision was made to develop a new grid middleware, JAliEn (Java ALICE Environment)[3]. The old grid middleware AliEn (ALICE Environment)[4] started displaying limitations when job queue length increased. Figure 1 shows that the number of jobs has increased drastically from the beginning of ALICE operations. There is also a steady increase in resources available on the grid. Together this creates a larger amount of updates to the job queue database, further straining AliEn. AliEn is also, at places, an amalgamation of quick fixes and hacks. As a product of this, continuing to develop AliEn to face future issues appearing is expensive and time-consuming. JAliEn was therefore introduced as a solution, removing some of the baggage

*e-mail: harn@hvl.no
[1]https://alice.cern/

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

that AliEn carried. As of today most of AliEn has been replaced by JAliEn, both centrally and on grid sites, with a few AliEn services running in the background that will be replaced in the coming time. One of these services that is still to be replaced is the job optimizer. The job optimizer service boils down to one thing, splitting jobs into subjobs if applicable. During this paper the term masterjob will refer to the original grid job submitted by a user, while subjobs of a masterjob is created by dividing the payload. Subjobs are executed in place of the masterjob. The focus of this paper is on the actual job splitting, and does not include other services that help facilitate this goal that is yet to be properly developed. The purpose of recreating a new job optimizer service is to create a more robust, scalable, and flexible implementation that hopefully will be better prepared for future challenges and more easily adapted to changes. Throughout this paper, some comparisons between the old and the new job optimizer will be done, and an introduction to the new job optimizer will be made.

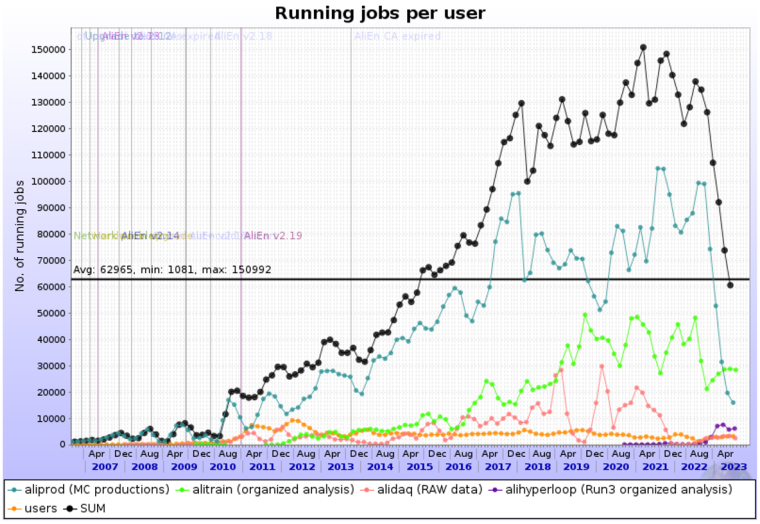


**Figure 1.** Number of running jobs per user, black being the summary of all

## 2 Job optimizer service architecture

One of the main goals of the new implementation of the job optimizer for JAliEn is for the user not to see any change to already existing workflows and how grid jobs are processed from a user's perspective. One minimum requirement is therefore as follows: "Grid jobs that worked for AliEn must work for JAliEn as well". However, in the background, several changes have been implemented to ease the load on some specific parts of the system, as well as ensure it adheres correctly to rules and limitations as one would expect. Old AliEn job optimizer services were a collection of several services, one being the job splitting, but other services such as the merging of subjobs were included under the job optimizer. Some changes have been introduced in JAliEn regarding which responsibilities fall under the job optimizer. As an example, merging of subjobs is no longer a part of the job optimizer. Figure 2 provides a rough flowchart of the job optimizer and technical parts that are involved.

Much like in AliEn, the job optimizer service in JAliEn wakes up periodically looking for grid jobs that are waiting to be split, but there is a fundamental change in the architecture. While both AliEn and JAliEn run several central machines that handle requests and services

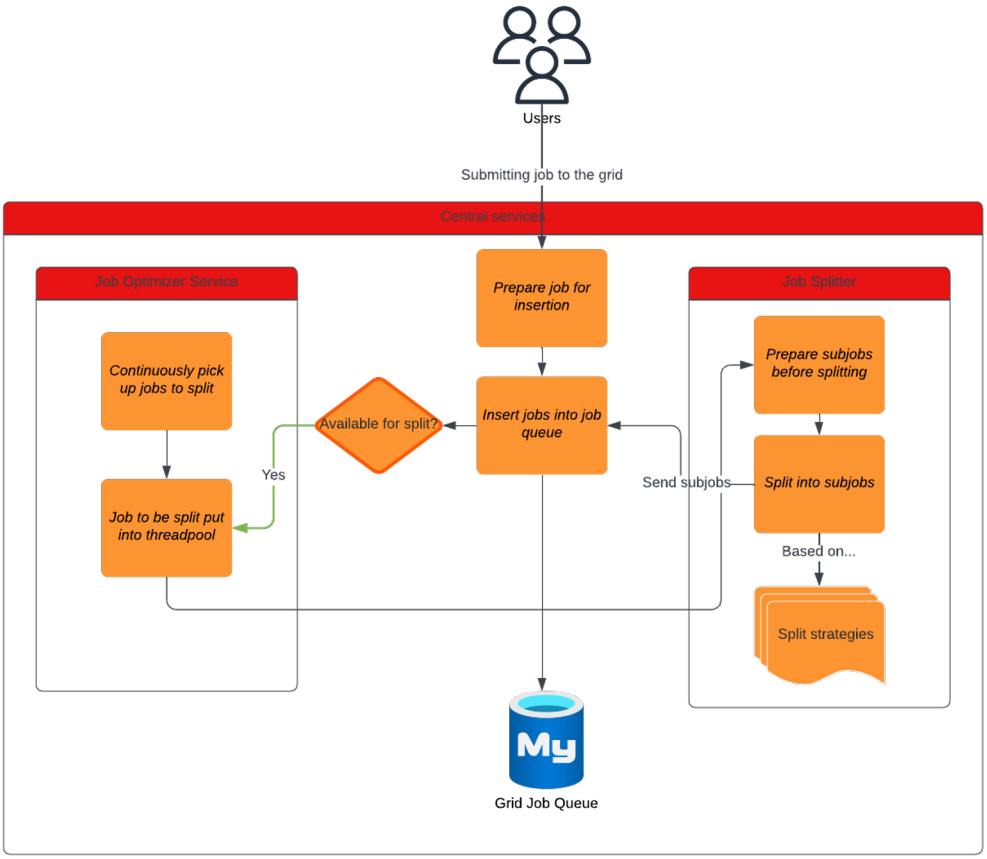


**Figure 2.** Flowchart of the job optimizer roughly detailed

for the grid to help distribute the load, the job optimizer was limited to a single machine in AliEn. This is not the case for JAliEn. Any machine running central services has the possibility of running the job optimizer service. Following the philosophy of JAliEn being more scalable and more configurable, a parameter can now be set in the configuration of a central service for the maximum number of threads in a thread pool that handles the splitting of a job, and the job optimizer is responsible for managing this thread-pool and starting a new thread in this pool. It is also possible to turn off the job optimizer service for any given central machine by setting the thread parameter to zero. This is the default value. In addition, a configurable thread pool helps with scalability, if there is a need to use more resources in the future for splitting, and it is cheap to increase or decrease resources used for this purpose.

## 3 Job optimizer service

As previously mentioned the job optimizer service in JAliEn is mainly focused on picking up grid jobs for splitting and insertion of subjobs. It has an internal queue in memory, the thread pool, that is responsible for starting the process of job splitting if there are available threads. In addition, the job optimizer has a custom handler, which triggers if the thread pool is full when trying to insert another thread. A central machine will always try to go directly to job

splitting when receiving a grid job to process and add to the grid queue. The benefit of this operation is removing the wait time for the periodic service to pick up the job at a later time. It will also increase the query effiency as there are few grid jobs to sort when picking a grid job for splitting. In the case where the thread pool is full, the custom handler will trigger and the grid job will be inserted for pick up at a later time by the periodic service. In addition to looking for jobs ready to be split the job optimizer will also look for grid jobs stuck in the job splitting state over a prolonged period. One such case could occur if a central machine shuts down while processing a grid job. Another central machine will instead pick the grid job up for processing. While rare, an issue is then introduced where it is a possibility to add duplicate subjobs if the first job splitting is still progressing, albeit very slowly. To remove this issue, insertion of subjobs in the database is synchronized by using a last updated value stored in the database. This value relates to the original masterjob, and if this value does not match it signals that the job was picked up for job split by another process or machine. All subjobs will then be discarded.

### 3.1 Job splitting

Splitting a larger grid job into smaller subjobs is done by splitting up the payload of the original grid job and assigning the different subsets of payload to different subjobs. It is not a division of the execution program. There is one exception to this which will be outlined later. Job splitting is done based on an explicit description provided in the Job Description Language (JDL), detailing how a job should be split as well as fields and patterns that are replaced during a job split for the subjobs. The patterns which are replaced are commonly used for counters in subjobs, or for getting a filename from the input. In AliEn these were limited to a subset of fields in the JDL, but this is not the case for the job splitting in JAliEn. Every field in the JDL is checked, but doing pattern matching is not cheap, and doing this individually for each subjob as it was done in AliEn, is not optimal. To combat this issue all pattern matching is done in advance, replacing the pattern with a lambda function. Using functional interfaces in combination with lambda functions it is possible to prepare all in advance, running the lambda functions after the job is split to get the final value. Since this matching is done on all JDL fields some fields are now redundant but are kept around just for the user's benefit, as one of the main goals is backward compatibility. The main reason to insert this complexity is to achieve a more flexible solution. One option this has opened up is the possibility of splitting a grid job into subjobs that each have a different execution program, by using counters to distinguish different programs.

#### *3.1.1 Strategies for a job split*

There are a few possible split strategies for splitting a job into subjobs, but can be grouped into the three categories outlined below.

- Production/Monte Carlo split
- Split by locality
- Split by logical filename

Production split differs from other splitting in that it does not distribute input data to different subjobs, but instead all subjobs retain all input data files. The subjobs within a production split do however use different seeds, which the counter dictates. It is possible to

set a range for the counter, getting the preferred seeds for the job. This is typically a Monte Carlo simulation.

Split by logical filename differs from splitting by locality in that the physical location of the files on the grid is not factored in. Instead input data becomes split based on its logical filename (LFN). Multiple options for this split exist, for example splitting based on filename or directory. Splitting based on locality looks at physical location on the grid, or physical filename (PFN), and not the LFN. For most of the different strategies of job splitting, the same number of subjobs will be created as the AliEn counterpart and will be mostly identical, the exception being for splitting based on locality.

JAliEn contains a central file catalogue that retains all records of PFNs for each LFN [5] and XrootD is used for data access and storage[6]. Sending grid jobs to be processed where the input data is stored removes the cost of having to transfer information over the network when reading the data on a different location. Therefore, splitting by locality is used to achieve this. Users submit grid jobs with the LFN and the job optimizer is responsible for translating over to physical location, and creating subjobs that take advantage of where input data is stored.

Splitting based on locality groups a set of input data files that shares the same set of physical location, called a basket. This creates unbalanced jobs as can be seen in Figure 3. The problem is twofold. Subjobs could be inefficiently large and taking up too much resources and time, or be so small that the overhead of processing the subjob is too large compared to the payload. To fix the problem of having some extremely large subjobs, it is mandatory to set a maximum number of files per subjob. It is mainly used when splitting by locality, but not limited to it. To face the problem of too small baskets, the new JAliEn job optimizer introduces the merging of smaller baskets to produce more balanced baskets and subjobs. To help achieve this merging, a new field that sets a minimum number of files per subjobs is also introduced, where the default value is one fifth of the maximum number of files per subjob. The merging itself happens mostly based on taking a smaller basket and merging it with a larger, but still under the threshold, basket where both have in common one or more physical locations. It is however not always the case where a basket might share a locality, and for such cases, a comparison of the distance between localities is made to figure out which baskets to merge. The localities within a basket are used as a requirement when matching grid jobs with grid sites, to ensure that data will not have to be moved long distances. Files on the grid with multiple physical locations are relatively close in distance. To take advantage of this, when balancing baskets only the larger basket's localities will remain. This is to try and avoid too much transfer of files when executing the job as only the smaller basket with the fewest files might have to be read from a remote storage, which is also relatively close.

## 4 Database optimization

One of the bottleneck areas in the central machines is the access to the database that is both keeping track of the grid queue, but also the catalogue which is responsible for keeping track of all files on the grid and physical location and more. While querying for information does not require many resources, even when several processes do it simultaneously, updating information and locking tables should be addressed and kept at a minimum. In addition, some changes had to be done to ensure limits related to users were followed. The old system did have checks related to quota for the number of grid jobs active and files on the grid, but it had some glaring issues. It was possible to go above the limit for the number of grid jobs,

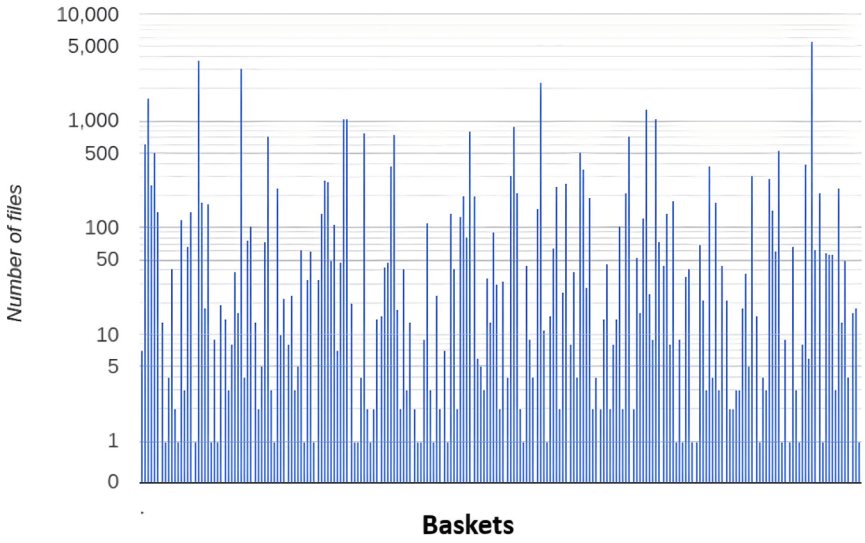


**Figure 3.** Input data files per basket for a split by location grid job

as it only checked if it was currently above, and if this was not the case the subjobs of a grid job would be inserted. To better define the limits set on users some changes have been made in this respect. Checks happen twice before all subjobs are inserted. Once when the original grid job was inserted, to ensure the quota was not already met, and once before all subjobs are inserted to ensure it would not go beyond the limits defined. In addition, to avoid the issue of inserting only a subset of all subjobs, all insertion queries happen within a transaction, which is another improvement from the old system. Either all subjobs or none is inserted.

### 4.1 Grid queue

When the job optimizer picks up a grid job to be split it will update the status, signaling that the grid job is currently in the process of being split. The job optimizer has to change the status of the grid job as well as return the job ID, which will be used later to retrieve the JDL of the given grid job. It is possible to use locking mechanisms and do a select query followed by an update query, but the actual table is already highly congested, and removing the need for locking is beneficial to improve efficiency as a whole. While some database systems provide the option of returning a value on an update query, this is not the case for the database used by the central services in JAliEn, which is MySQL. The solution is then to use user-defined variables[7]. During an update, it is possible to set a user-defined variable in the database, which later can be retrieved through a select query that does not target the active queue table, and therefore no need to lock anything after the update query is done. The only issue is that it is not possible to pick up several job IDs for splitting in a single query, and the user-defined variable can only contain one value, not a list of values. This means that if an update query on several rows occurs, only the job ID of the last row will be available in the user-defined variable. So in the cases where the job optimizer wants to start splitting more than one grid job using different threads, as mentioned earlier, it has to do an update query individually to acquire each job ID.

All jobs and subjobs have a corresponding JDL stored in the database as a string, taking up quite a bit of resources. AliEn stored full JDLs for all jobs and subjobs, but inspecting the JDLs reveals large part are redundant and duplicated between subjobs. To save storage resources a change to how JDLs are described and stored for subjobs have been implemented. There are no changes to JDL storage for the original masterjob, but subjobs are now stored as Delta of masterjob JDL, removing all redundant information that is already detailed in the masterjob.

### 4.2 Catalogue

The other part of communicating with the database is specifically tied with splitting based on file locality, as other split strategies are only looking at the logical file path, which is already provided in the collection. Each file has a virtual file path on the grid, but to get the physical localization on the grid it is necessary to query the database twice. This becomes an issue for a large collection, that takes up both time and resources from other services, but can also be slow. A solution to this is to make use of the database structure used to store all relevant file metadata as an advantage to perform checkups as a bulk operation, but it is not a single table that contains the information for all files. This table is partitioned into many smaller tables, each having its unique table name. Taking advantage of this feature it is possible to group files based on which partition and the table name the file's metadata is located on, then do a single query for all files contained within a unique table name, instead of doing an individual query for each file in a collection. This helps with freeing up resources for the database, as well as faster and more efficient retrieval of the physical location of a file. This improvement is compounded by the fact that this process has to happen twice to go from logical to physical location as a bare minimum, but could occur up to four times.

## 5 Summary

Run 3 comes with larger demands of efficiency and performance with the increase in data being produced. To meet this demand the new job optimizer was created, with a bigger focus on improving scalability and making it more adaptable for the future. An effort has been made to improve parts that were not flexible, or not following limitations set, such as quotas. Changes to the architecture of the job optimizer, specifically allowing all central machines to run the job optimizer, have increased the load of traffic the job optimizer can serve, and added more scaling if necessary. Balancing of subjobs regarding locality splitting has seen a decrease in the number of subjobs with too few input data files, but a larger test of the full impact of the balancing of baskets remains.

Improving communication towards the database has been important, as this is currently a bottleneck and creates performance issues that ripple throughout JAliEn. Burden on the database storage resource has also been lessened as storing subjobs takes less space.

There is still work to be done regarding services closely related to job splitting, which could also improve some of the inefficiency problems related to the locking of tables in the database, and is something that will be developed going forward.

## References

[1] D. Colella (ALICE), Int. J. Mod. Phys. E **31**, 2240002 (2022), 2206.13247
[2] L.V. Palomo, Journal of Physics: Conference Series **912**, 012023 (2017)

[3] A.G. Grigoras, C. Grigoras, M.M. Pedreira, P. Saiz, S. Schreiner, Journal of Physics: Conference Series **523**, 012010 (2014)

[4] S. Bagnasco, L. Betev, P. Buncic, F. Carminati, C. Cirstoiu, C. Grigoras, A. Hayrapetyan, A. Harutyunyan, A.J. Peters, P. Saiz, J. Phys. Conf. Ser. **119**, 062012 (2008)

[5] P. Cortese, F. Carminati, C.W. Fabjan, L. Riccati, H. de Groot (ALICE), *ALICE computing: Technical Design Report*, Technical design report. ALICE (CERN, Geneva, 2005), submitted on 15 Jun 2005, `https://cds.cern.ch/record/832753`

[6] A. Dorigo, P. Elmer, F. Furano, A. Hanushevsky, WSEAS Transactions on Computers **1**, 348 (2005)

[7] *Mysql user defined variable*, `https://dev.mysql.com/doc/refman/8.0/en/user-variables.html`, accessed: 21-08-2023