Demo: SLASH: Serverless Apache Spark Hub

Josef Spillner
josef.spillner@zhaw.ch
Zurich University of Applied Sciences
Winterthur, Switzerland

ABSTRACT

Application needs for big data processing are shifting from planned batch processing to emergent scenarios involving high elasticity. Consequently, for many organisations managing private or public cloud resources it is no longer wise to pre-provision big data frameworks over large fixed-size clusters. Instead, they are looking forward to on-demand provisioning of those frameworks in the same way that the underlying compute resources such as virtual machines or containers can already be instantiated on demand today. Yet many big data frameworks, including the widely used Apache Spark, do not sandwich well in between underlying resource managers and user requests. With SLASH, we introduce a light-weight *serverless* provisioning model for worker nodes in standalone Spark clusters that help organisations slashing operating costs while providing greater flexibility and comfort to their users and more sustainable operations based on a unique *triple scaling* method.

CCS CONCEPTS

• **Computing methodologies** → *Distributed algorithms*; • **Networks** → *Network services*; • **Computer systems organization** → **Maintainability and maintenance**.

KEYWORDS

Autoscaling, big data, provisioning

ACM Reference Format:
Josef Spillner. 2023. Demo: SLASH: Serverless Apache Spark Hub. In *The 17th ACM International Conference on Distributed and Event-based Systems (DEBS '23), June 27–30, 2023, Neuchatel, Switzerland.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3583678.3603277

DEBS '23, June 27–30, 2023, Neuchatel, Switzerland

ACM ISBN 979-8-4007-0122-1/23/06.
https://doi.org/10.1145/3583678.3603277

1 INTRODUCTION

Scalable data pipelines with storage and processing capabilities are an indispensible ingredient for most application architectures. Ranging from business number crunching and machine learning to nation-scale handling of people, services or things interactions, the variety of data management approaches has widened considerably in recent years [1]. Apart from conventional database management systems, including flavours such as NoSQL, in-memory and key-value stores, several big data processing frameworks, often with tangential handling of long-term data storage, have emerged. Among the most well-known frameworks are Hadoop, Spark, Flink, Ray and Dask. Most of these frameworks have turned into industrial de-facto standards, underlined by vendor-neutral open source communities such as the Apache Foundation.

Key advantage of these open frameworks include their portability and extensibility. While they often predate modern resource managers and are thus harder to set up than recent cloud-native data management and processing offerings, they provide a predictable interface that remains the same even when migrating from on-premise deployments to the cloud or between cloud providers. Individual deployments can still be customised. These and other advantages are well understood and covered by the literature [4, 12].

However, in contrast to recent approaches such as fully serverless data processing approaches like with Lithops, the hurdle on setting up such pipelines on static clusters remains. Hence, with SLASH, a path towards better on-demand self-service, inspired by a serverless mindset, is shaped while at the same time providing a way for organisations to protect their investments into existing big data setups. Specifically, SLASH offers a way to deploy and invoke Spark workers as dynamic middleware along with Spark-based applications, based on *triple scaling*: fixed schedules, anticipated usage, and on-demand. In contrast to commercial Spark offerings such as Databricks, users no longer need to switch on and off clusters.

In the next section, the challenges behind this approach are presented. In the remaining sections, the SLASH system architecture is described and the implementation is demonstrated in production in a research and education cloud infrastructure.

2 CHALLENGES

Bounded elasticity defines the ability of a system to follow a load curve while adhering to other constraints, such as maximum availability of resources and minimum idleness. These sizing constraints may be economically motivated but also have a profound effect on the environmental footprint of a system. When a system underscales the load curve, its users may become dissatisfied with higher makespans so that there is a trade-off between economic/ecologic and social/psychologic concerns.

On a technical level, bounded load-proportional processing in any big data framework is achieved by horizontal and vertical scaling. An auxiliary scaling framework takes care of allocating resources as part of general resource management. The knowledge for taking proper scaling decisions may come from different sources: explicit schedules (calendar-based predictive autoscaling), implicit schedules (forecasting based on historic behaviour), and on-demand requests (reactive autoscaling) which are well-supported in modern resource orchestrators such as Kubernetes. The first challenge is combining these sources in real-time and deriving right-sized application instances from the estimated or calculated overall resource requirements and constraints through a *triple scaling* method. Kubernetes in particular is aimed at short-lived *cattle* workloads, whereas Spark contexts do not tolerate resource variability during execution, and moreover provides only laggy resource metrics access; hence, even in containerised environment a custom autoscaler approach is needed. The second challenge is conveying this method to the application engineer without the need to manage servers, following the seemingly *serverless* approach that has become popular in industry with cloud applications based on FaaS. Recent big data frameworks such as Lithops [9] already follow such a serverless design, while this work intends to combine it with triple scaling and apply it to a legacy static cluster framework such as Spark.

3 SYSTEM ARCHITECTURE

SLASH is based on a distributed event-based architecture with a central hub receiving on-demand events from applications as well as regular events from its own scheduled jobs. It also queries the underlying resource manager (hypervisor) and the Spark master to be informed about actual resource usage, including from non-autoscaled workloads that may coexist and need to be factored in. The hub can be deployed on the Spark master node or on a dedicated node with equal reachability from applications. Due to its ability to collocate with non-autoscaled workloads, it can also run on the application or frontend nodes, such as Jupyter notebooks. A collocated job server process ensures that regular calendar events, forecasting events and requests for a guaranteed base capacity are all forwarded to the hub at the appropriate times, which then merges these with on-demand requests to decide on scale-up/-down instructions to the resource manager. Fig. 1 shows how the SLASH components (hub, jobserver, application proxy class) fit into a typical Spark deployment.

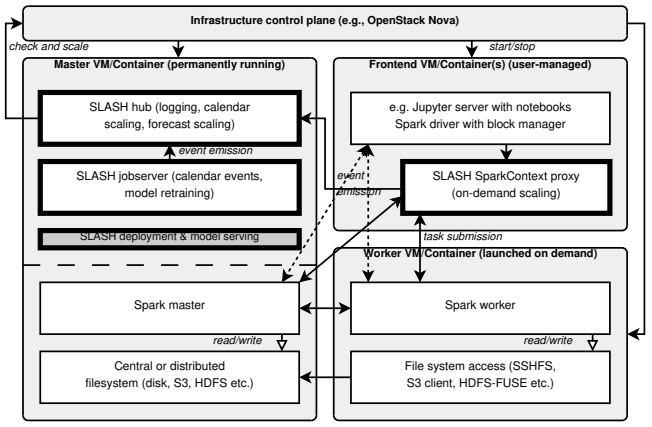

Figure 1: Portable SLASH system architecture

For the big data application engineer, making use of SLASH is trivial. Listing 3 gives a practical example of how to incorporate SLASH on-demand scheduling with a single line of code into typical Spark applications. The module import only relies on an environment variable `$SLASH` pointing to the hub (`<ip>:<port>`) that must be pre-set in the environment (e.g. Jupyter notebook) or programmatically inside the application code.

```
import pyspark
...
import slash # auto-upgrade pyspark to SLASH

sc = pyspark.Context(...)
# scale up here latest unless pre-scaled already
sc.parallelize([...])
sc.stop()
# scale down here if applicable unless overridden
    by calendar/forecasting/basecap
```

Each application produces a custom events track which can be analysed for e.g. cleaning up aborted applications not sending the disconnection events. SLASH consolidates all received events on all tracks and merges them into an authoritative decision track in order to avoid conflicting upscaling/downscaling decisions. The decision track is the input to the infrastructure scaling layer. This interaction is implemented for vanilla OpenStack, but could also be extended to scientific compute environments such as Chameleon Cloud[1], and to container orchestrators such as Kubernetes for which Spark integration is available[2]. Fig. 2 explains the cumulative

[1]Chameleon: https://www.chameleoncloud.org/

[2]see https://spark.apache.org/docs/latest/running-on-kubernetes.html

nature of the tracks over a hypothetical timeline. Each track contains a number of cores at each point in time, adding up to the overall allocation. The figure omits the base capacity track and the number of cores.

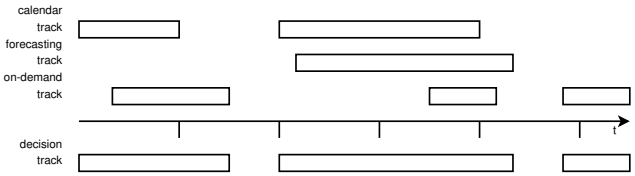

Figure 2: Cumulation of scaling event tracks into scaling decisions

Careful downscaling happens either explicitly based on follow-up events, or implicitly based on a defined timeout, e.g. 4 hours for classroom settings. The careful behaviour leaves still allocated resources untouched. The appropriate way to de-allocate them would be a timeout or enforced termination on the application itself, rather than the autoscaler.

4 SYSTEM DEMONSTRATION

Zurich University of Applied Sciences runs a 96 cores Spark cluster for educational and research/industry innovation purposes, primarily for data science and big data applications with streaming (ESP/CEP), graph processing, and machine learning (ML). We have deployed SLASH on this infrastructure and tested it with 30 concurrent users to demonstrate its ability to integrate, scale, and cut down resource consumption. The underlying infrastructure is a full deployment of OpenStack 23.01 (Antelope) with public resource description[3]. All Spark-related VMs run Ubuntu 23.04 (Lunar Lobster) with Python 3.11 and Spark 3.4.0.

The open source SLASH system [10] has been deployed on one of the shared collaborative Jupyter notebook environments attached to the Spark cluster. This environment is used in production and therefore allows collecting real usage statistics.

Fig. 3 shows a synthetic session where blue requests for cores trigger orange allocation of virtual machines with multiples of 8 as available cores. Downscaling is delayed to avoid throttle, as can be seen in the second orange section.

Fig. 4 shows a trace of a 90 minutes classroom session. First, autoscaling was deactivated, leading to Spark applications remaining in `WAITING` mode due to insufficient resources (blue line over orange line). Then, it was enabled, and immediately provisioned the necessary 48 cores. Afterwards, classroom mode with full provisioning for that day with 64 cores was activated, which prevented downscaling even in cases of mass application terminations, as can be seen by the two

[3]Cloudlab: https://info.cloudlab.zhaw.ch/

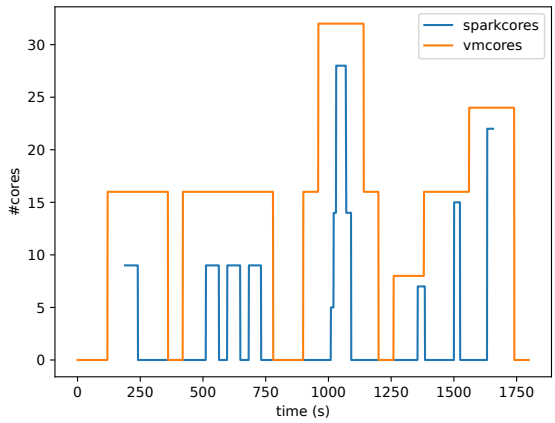

Figure 3: Measured autoscaling behaviour over time: VM cores versus effectively used Spark cores

OOM-related Jupyter crashes bringing the resource demand temporarily down to 0, and thus prevented student anger over more waiting time after reconnecting their notebooks.

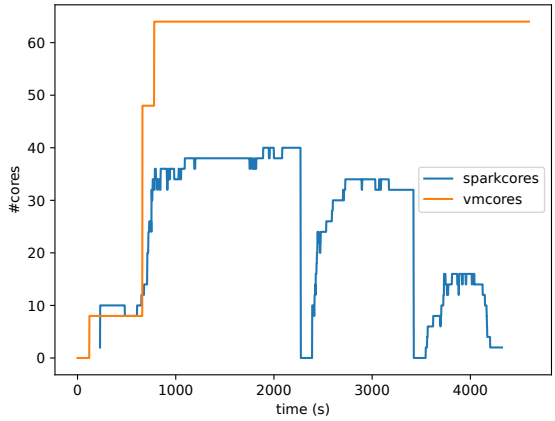

Figure 4: Autoscaling behaviour in a classroom setting

SLASH has been used for several dynamic data processing applications, in various embeddings such as Jupyter notebooks, batch jobs and interactive desktop visualisations. One of the latter types is a live event stream analysis of delays, cancellations and other GTFS-RT matters on the Swiss public transport system, based on Spark Streaming. Fig. 5 conveys the need to scale when considering an influx of events scattered across administrative regions. Due to a weekly backlog of the data in the original source, this is especially required after application startup with hundreds of events/s.

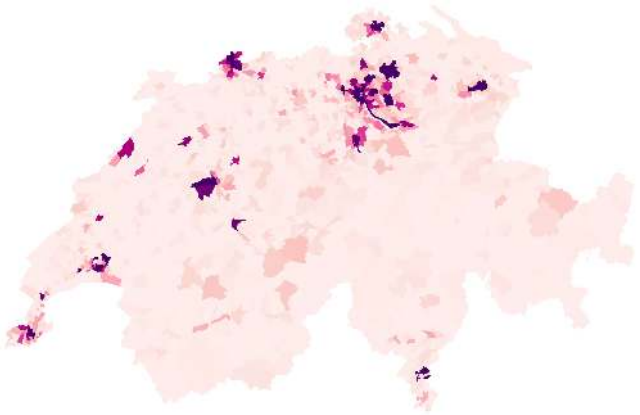

Figure 5: Horizontally scaled stream analysis on country-wide public transport events

5 RELATED WORK

Spark has been a popular big data processing and analytics framework for over a decade [5], based on a master-worker model and application contexts occupying resources statically on workers during execution. It is often used as basis for other systems such as distributed geospatial pub/sub brokers [7] due to its ability to exploit existing resources with integrated resource management. However, the resources are treated as static allocations, and scaling decisions are not propagated downward to the underlying resource manager. More dynamic resource management for big data frameworks has therefore become a research direction [6]. Dynamic resource provisioning and scaling for Spark in particular has led to a number of research results over almost a decade [2, 3]. Among the more recent results is a performance study of autoscaling Spark atop Kubernetes [11] which confirms dynamic executor allocation on worker nodes (as pods) but also that the worker nodes themselves remain static (setting `spark.dynamicAllocation.enabled`). Another approach for scaling Spark on top of OpenStack is based on SLAs with specified deadlines [8]. All of these approaches are bound to the runtime environment, such as Spark hosted on IaaS, CaaS or HPC. SLASH is similarly bound, currently to OpenStack. But what differentiates it is its leverage of the seemingly serverless approach. It therefore makes the scaling across infrastructure layers controllable by the user. Moreover, SLASH combines several event sources to accommodate all scaling needs, and delivers information usually not accessible from PySpark, such as the number of effectively allocated cores following the requested number.

6 CONCLUSIONS

SLASH makes Spark worker scaling more elastic based on a *triple scaling* method. The system has been shown to work and perform well in a reasonably sized production environment. From an operations perspective, SLASH helps slashing resource operation cost, whereas from a user perspective, it conveys a seemingly serverless interface to launching workers as needed up to the operator-defined limits. SLASH is available as open source implementation [10] and can be considered for adoption by operators of Spark environments with different frontends. Future work will focus on support for other frameworks so that their workers can co-exist on the same virtual machines, based on the introduction of additional event tracks in the hub, as well as on more useful forecasting based on seasonalities such as course weeks.

REFERENCES

[1] Pouya Ataei and Alan T. Litchfield. 2022. The State of Big Data Reference Architectures: A Systematic Literature Review. *IEEE Access* 10 (2022), 113789–113807. https://doi.org/10.1109/ACCESS.2022.3217557

[2] Nicholas Chaimov, Allen D. Malony, Shane Canon, Costin Iancu, Khaled Z. Ibrahim, and Jay Srinivasan. 2016. Scaling Spark on HPC Systems. In *Proc. 25th ACM Intl. Symp. High-Performance Parallel and Distributed Computing, HPDC 2016, Kyoto, Japan, May 31 - June 04, 2016*. ACM, 97–110. https://doi.org/10.1145/2907294.2907310

[3] Dazhao Cheng, Yu Wang, and Dong Dai. 2023. Dynamic Resource Provisioning for Iterative Workloads on Apache Spark. *IEEE Trans. Cloud Comput.* 11, 1 (2023), 639–652. https://doi.org/10.1109/TCC.2021.3108043

[4] Christos Doulkeridis, Akrivi Vlachou, Nikos Pelekis, and Yannis Theodoridis. 2021. A Survey on Big Data Processing Frameworks for Mobility Analytics. *SIGMOD Rec.* 50, 2 (2021), 18–29. https://doi.org/10.1145/3484622.3484626

[5] Diego García-Gil, Sergio Ramírez-Gallego, Salvador García, and Francisco Herrera. 2017. A comparison on scalability for batch big data processing on Apache Spark and Apache Flink. *Big Data Analytics* 2, 1 (01 Mar 2017), 1. https://doi.org/10.1186/s41044-016-0020-2

[6] Athanasios Kiourtis, Panagiotis Karamolegkos, Andreas Karabetian, Konstantinos Voulgaris, Yannis Poulakis, Argyro Mavrogiorgou, and Dimosthenis Kyriazis. 2022. An Autoscaling Platform Supporting Graph Data Modelling Big Data Analytics. In *ICIMTH 2022*, Vol. 295. IOS Press, 376–379. https://doi.org/10.3233/SHTI220743

[7] Ivan Livaja, Krešimir Pripužić, Siniša Sovilj, and Marin Vuković. 2022. A distributed geospatial publish/subscribe system on Apache Spark. *Future Generation Computer Systems* 132 (2022), 282–298.

[8] Yoori Oh, Jieun Choi, Eunjung Song, Moonji Kim, and Yoonhee Kim. 2016. A SLA-based Spark cluster scaling method in cloud environment. In *2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. 1–4. https://doi.org/10.1109/APNOMS.2016.7737242

[9] Josep Sampé, Marc Sánchez Artigas, Gil Vernik, Ido Yehekzel, and Pedro García López. 2023. Outsourcing Data Processing Jobs With Lithops. *IEEE Trans. Cloud Comput.* 11, 1 (2023), 1026–1037. https://doi.org/10.1109/TCC.2021.3129000

[10] Josef Spillner. 2023. *SLASH - Serverless Apache Spark Hub*. https://doi.org/10.5281/zenodo.7897070

[11] Vinay Kumar Vennu and Sai Ram Yepuru. 2022. A performance study for autoscaling big data analytics containerized applications: Scalability of Apache Spark on Kubernetes. urn:nbn:se:bth-22685.

[12] Xin Wang, Pei Guo, Xingyan Li, Jianwu Wang, Aryya Gangopadhyay, Carl E. Busart, and Jade Freeman. 2021. Reproducible and Portable Big Data Analytics in the Cloud. *CoRR* abs/2112.09762 (2021). arXiv:2112.09762 https://arxiv.org/abs/2112.09762