Skadi: Building a Distributed Runtime for Data Systems in Disaggregated Data Centers

Cunchen Hu[1,2]*, Chenxi Wang[1,2], Sa Wang[1,2], Ninghui Sun[1,2], Yungang Bao[1,2], Jieru Zhao[3], Sanidhya Kashyap[4], Pengfei Zuo[5], Xusheng Chen[5], Liangliang Xu[5], Qin Zhang[5], Hao Feng[5], Yizhou Shan[5]

[1]University of Chinese Academy of Sciences, [2]State Key Lab of Processors, ICT, CAS
[3]Shanghai Jiao Tong University, [4]EPFL, [5]Huawei Cloud

ABSTRACT

Data-intensive systems are the backbone of today's computing and are responsible for shaping data centers. Over the years, cloud providers have relied on three principles to maintain cost-effective data systems: use disaggregation to decouple scaling, use domain-specific computing to battle waning laws, and use serverless to lower costs. Although they work well individually, they fail to work in harmony: an issue amplified by emerging data system workloads.

In this paper, we envision a distributed runtime to mitigate current shortcomings. The distributed runtime has a tiered access layer exposing declarative APIs, underpinned by a stateful serverless runtime with a distributed task execution model. It will be the narrow waist between data systems and hardware. Users are oblivious to data location, concurrency, disaggregation style, or even the hardware to do the computing. The underlying stateful serverless runtime transparently evolves with novel data-center architectures, such as disaggregation and tightly-coupled clusters. We prototype Skadi to showcase that the distributed runtime is practical.

ACM Reference Format:
Cunchen Hu, Chenxi Wang, Sa Wang, Ninghui Sun, Yungang Bao, Jieru Zhao, Sanidhya Kashyap, Pengfei Zuo, Xusheng Chen, Liangliang Xu, Qin Zhang, Hao Feng, Yizhou Shan. 2023. Skadi: Building a Distributed Runtime for Data Systems in Disaggregated Data Centers. In *Workshop on Hot Topics in Operating Systems (HotOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3593856.3595897

1 INTRODUCTION

Motivation. Today's cloud has been deeply shaped by data-intensive systems since the early 2000s when GFS [22] was built to cope with the boom of big data. The volume of data and the variety of data-intensive systems in the cloud increased exponentially in the past two decades [1]. Many data-intensive systems migrate to the cloud for instant infrastructure availability.

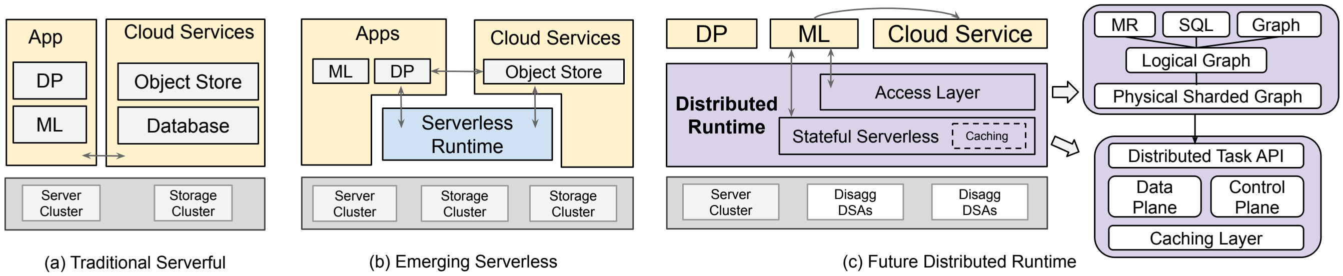
Nevertheless, it is incredibly challenging to design, deploy, and run data systems fast enough at a reasonable cost, as they commonly use deep storage and can be both compute-intensive and memory-intensive [12, 67]. Over the years, users, data system designers, and cloud vendors have worked closely to keep running data systems cost-effective using three principles: (1) use disaggregation to enable independent resource scaling; (2) use domain-specific accelerators to battle waning laws and improve computing efficiency; (3) use serverless to lower costs and boost productivity.

These principles work well individually, but they fail to work in harmony as each was proposed to solve a different problem (explained next), an issue amplified by two recent trends. The first trend is data systems integration in which multiple data systems are deployed onto one pipeline that jointly runs business logic, data management, HPC, and ML [1, 4, 14, 26, 40, 46]. For example, BigQuery can run data ingestion, extraction, analytics, and ML in one job [26]. The second trend is that giant model training has evolved from using SPMD to MPMD over multiple highly-specialized clusters [7, 12, 79]. Both workloads are commonly written in high-level, declarative languages (e.g., SQL), and developers of these workloads are mostly oblivious to how the underlying computation is carried out. For the best cluster

*Work done while intern at Huawei Cloud.

HotOS '23, June 22–24, 2023, Providence, RI, USA

ACM ISBN 979-8-4007-0195-5/23/06.
https://doi.org/10.1145/3593856.3595897



(a) Traditional Serverful

(b) Emerging Serverless

(c) Future Distributed Runtime

Figure 1: Towards a distributed runtime. *In (a), both user data systems and cloud services run on a logically-disaggregated cluster using regular servers (e.g., cloud storage on the right is disaggregated from compute on the left). In (b), both user data systems and cloud services partially migrate to stateless serverless. But some logic is still running on regular servers provisioned separately. In (c), we envision most data systems and cloud services would run on top of the distributed runtime. DP is short for data processing systems.*

cost-efficiency, both trends call for a close co-design of programming models, runtime, networking, data-center hardware, etc. To motivate this paper, we now discuss what the above principles are, how they conflict, and why they fail to accommodate emerging workloads.

First, cloud vendors adopt the disaggregation principle at two levels in unison for a higher cluster efficiency. At the *logical* level, vendors build separate resource pools using regular servers [6, 44, 58, 67]. This decouples compute from storage and enables each to scale independently to best fit data system needs. However, the unit for scaling and running tasks is limited by the rigid server box [59]. *Physical* disaggregation emerged to break servers into customized devices, which often consist of a dominant resource like DRAM or GPU along with ancillaries like DPU for networking and control [28, 37, 45, 59, 60]. It can greatly improve resource utilization. To date, co-designing data systems with logical-disaggregation is battle-tested [6, 13, 44, 65, 67, 70]. But adapting data systems with physical-disaggregation is still in its infancy [21, 37, 78].

The second principle is that vendors seek domain-specific accelerators (DSA) to overcome limitations imposed by conventional computing power such as CPUs [30] to keep up with data systems' increasing computing need [6, 10, 18, 33]. Unlike CPUs, DSAs have intrinsically different usage models. Hence using a high-level framework [19, 53] is crucial to reap their benefits. However, integrating DSAs with data systems is a demanding but mostly one-time effort [6, 18, 33, 42, 56], often creating "computing silos" in which DSAs are exclusively owned by a data system or a service [6, 32]. Such computing silos can be tightly-coupled clusters in which DSAs are interconnected via high-speed interconnect, essentially trading the scale of the cluster for the best performance. This can result in suboptimal cluster utilization, which conflicts with the disaggregation and pooling principle. It also makes sharing DSAs across distinct data systems more difficult as the DSAs are usually not exposed as-is but with added-values.

Finally, the serverless principle further lowers cost by offering a pay-as-you-go cost model over a reservation-based one and boosts productivity by hiding distributed clusters. However, existing commercial serverless lacks key features that hinder its broad adoption [57]. In particular, compute and storage are separated due to logical disaggregation. As a result, functions usually bounce data via durable cloud storage [36, 55]. Unfortunately, this is detrimental to data systems that heavily rely on a fast caching layer for storing states and ephemeral data exchanged across functions. This issue is more prominent with data systems integration. In addition, existing serverless uses a CPU-centric model in which users run general-purpose code on CPUs and orchestrate DSAs from there, and the auto-scaling of DSAs is almost non-existent [54]. Due to these limitations, users and even cloud services only partially migrate to commercial serverless and still exchange data via slow durable storage, as reflected in Figure 1b.

We believe the current ecosystem is not optimal for modern workloads written in high-level, declarative languages and running multiple demanding data systems in one pipeline. Ideally, we need a solution that simultaneously provides all the benefits enabled by the above three principles. The said solution must meet the following requirements: (a) has an easy programming model that enjoys the pay-as-you-go model for *all* the computing power used, (b) optimizes for cluster cost-efficiency such as reducing data movement whenever possible, and (c) transparently evolves with data-center infrastructure. In this paper, we propose a distributed runtime to close the gap.

Our work. We envision that the **distributed runtime** has a tiered access layer that exposes declarative data-centric and functionality-centric APIs, underpinned by a stateful serverless runtime with a distributed task execution model and flexible state management. It is the narrow waist between data systems and hardware (Figure 1c). As such, it achieves the separation of concerns at scale: users apply domain-specific declarative computation on data, oblivious of data location, concurrency, consistency, disaggregation style, or even the hardware used to do the compute. The tiered access layer will map declarations onto a universal sharded graph that dictates how data flow through computing tasks. On the other hand, the underpinning stateful serverless runtime will run the graph and ensure efficient task scheduling and data movement, all the while transparently evolving with novel data-center architecture such as disaggregation [59] and tightly-coupled clusters [23, 32].

We believe it will be in the cloud vendors' best interest to exploit the co-design of data systems, the distributed runtime, and the data center infrastructure in order to achieve the best cost-efficiency. As such, more user data systems and cloud services would migrate to it. This necessitates the distributed runtime to host data systems with varied execution models such as BSP [16, 31, 44, 75], task-parallel [35, 46, 49], streaming [48, 76], graph [24, 25, 50], ML [2, 7, 11, 79], etc. A flexible access layer and a stateful serverless runtime are key to realizing this.

The **access layer** is proposed based on decades of research and industry best practices on data flow. It consolidates fragmented domain-specific declarations onto one execution graph. It does so using three tiers: a domain-specific declarative tier, a logical graph tier, and a physical sharded graph tier. Domain-specific ones include SQL [5], Graph [24], MapReduce [16], ML [2, 11], etc. They are collectively lowered onto one logical graph that hosts data-parallel, task-parallel, and iterative patterns at once, using hardware-agnostic vertices connected by directed edges. Lowering a logical graph to a physical graph means possibly creating sharded vertices along keyed edges and then mapping vertices to hardware operators. Both the generated logical and physical graph dictate *how* data flows through vertices. Crucially, they do not specify *when* and *who* should execute the vertices, a task delegated to Skadi's stateful serverless runtime.

The tiering model mirrors Dryad [31, 73] and Naiad [48], but differs in how vertices are built. We propose to use IR-based primitives, in addition to predefined operators (e.g., handcraft kernels [31, 75] or wrapped utilities [34, 48]) to build the vertices in the logical graph, akin to [14, 34]. A common IR enables graph-level optimizations such as op-fusing *across* application domains, in contrast to being confined within one domain [11]. This IR is data-centric, focusing on expressing the flow of data and the scheduling of compute. It is also functionality-centric: distributed runtime developers focus on functionalities intended, free from selecting hardware or porting the same operator to multiple hardware platforms, a task complicated by the emerging disaggregated [28, 37] and heterogeneous devices [8, 10, 33, 56] with diverging hardware characteristics.

The **stateful serverless runtime** will execute the physical graph from the access layer. It resembles a task-parallel system with a universal dynamic task execution API [35, 46, 49], based on which many distributed computing patterns can be built, e.g., generic data-parallel and task-parallel patterns, or the specialized MPMD pattern in giant model training [7]. The stateful serverless runtime has a flexible control plane and a fast data plane.

Its control plane is responsible for resource management, task dispatching, auto-scaling, etc. For workloads with long-running operators (e.g., analytics), it has a minor impact on performance but determines utilization. For workloads with frequent short operators (e.g., ML), it determines performance. To this end, the control plane embraces data-centric scheduling [27] for higher utilization, and forgoes the CPU-centric model to better support short-lived operators on heterogeneous hardware. If necessary, it could also integrate gang-scheduling to support SPMD-style sub-graph [7].

The stateful serverless runtime's data plane is critical as data systems normally move a sizable amount of data to finish a job, reflected in the data size per transfer, the cost paid per transfer, and the number of transfers. One can certainly use direct hardware channels [15, 52] or novel networks [23, 43] for good performance. Above all, we believe a fast caching layer with a standard format [3] is the bedrock of our data plane. It enables stateful functions because it can store states, external storage's input/output, and ephemeral results exchanged by functions within a job or across systems. Crucially, it has four benefits: (1) It decouples compute from states so compute (i.e., vertices) can be opportunistically migrated to where data reside to reduce data transfer. (2) A shared format such as Arrow [3] enables functions running on heterogeneous devices to exchange data without costly data marshalling, hence reducing the cost paid per transfer. (3) It unties data systems within an integrated pipeline using futures [7, 46, 63], thus enabling pipeline parallelism across system boundaries. Also, it can reduce the number of trips to durable storage. (4) An optional highly-available caching can replace lineage [46], adding another design trade-off.

We prototype **Skadi** to showcase that the distributed runtime vision is practical. For the access layer, we build *FlowGraph* as the logical graph. Its vertices use a multi-level IR [41] to express hardware-agnostic computation. Skadi will lower the logical graph to a physical one. The physical graph is a set of linked Python objects in our current implementation. Skadi will parse the graph and then launch tasks accordingly using stateful serverless runtime's task API. Skadi's stateful serverless runtime is built based on Ray. We first offload Ray's control and data plane to DPUs and extend its ownership table to enable Ray on physically-disaggregated devices. To support short-lived ops, we eschew Ray's CPU-centric model and offload part of its control plane (i.e., raylet) to heterogeneous devices. In addition, we add another push-based future resolution scheme.

2 DESIGN AND RESEARCH AGENDA

We prototype Skadi to demonstrate that the distributed runtime idea is viable. This section will discuss Skadi’s initial design, research agenda, and open questions.

2.1 Skadi Overview

We present Skadi’s architecture in Figure 2. Skadi enables users to use only one runtime to express all of their programs as a part of this runtime. The input consists of several domain-specific declarations like SQL statements and ML training (e.g., a python script). Skadi maps the program onto a logical graph in two steps: (1) invokes domain-specific parsers to translate declarations onto a common graph called FlowGraph, whose edges dictate how data flow and vertices are built with either handcraft operators (ops in short) or hardware-agnostic ops using MLIR [41]; (2) optimizes the graph using predefined rules.

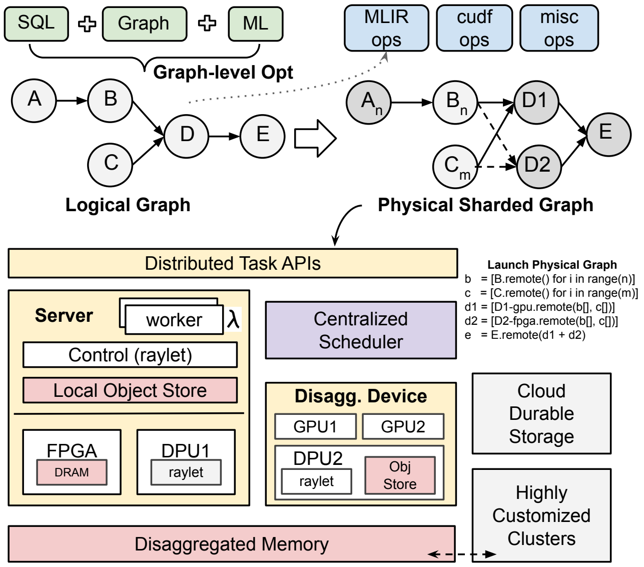
Skadi lowers the logical FlowGraph to a physical sharded graph in two steps: (1) selects hardware backends for MLIR-based ops using predefined rules; (b) decides a default degree of parallelism for each vertex (subscripts in Figure 2), and keyed edges with a default or user-supplied hashing scheme.

Skadi launches functions according to a given physical graph using stateful serverless runtime’s task APIs (pseudo-code in Figure 2). Functions exchange data either by value or by reference. Before scheduling a function, the runtime decides the preferred hardware based on memory locality, device availability, network topology, etc. Once dispatched, the function either runs immediately regardless of whether the input is available or not, or waits until the input is ready. The wait mode is possible because (a) Skadi’s control plane supports data-centric scheduling [27] and (b) Skadi supports pass-by-reference (i.e., futures) and has a flexible caching layer. The caching layer has a simple KV API for memory on regular servers, memory on heterogeneous devices, and disaggregated memory. Crucially, the caching layer can hide the location and movement of data.

Skadi handles failures in two ways: (1) re-executes the graph using lineage [46], or (2) uses a reliable caching layer with data replication [61] or EC [80]. Most existing data systems use lineage since replication is costly [35, 46, 49, 75]. However, a reliable caching layer could be beneficial as it helps reduce tail latency and potentially cost since the cost of restarting jobs may offset the cost of extra storage.

2.2 Access Layer

The access layer, as its name suggests, grants data systems access to the distributed runtime. Hence its top-level APIs must be compatible with existing ecosystems. Fortunately, it can reuse mature and open-source definitions, tools, and compilers [14, 19, 34, 53, 75]. For example, FlowGraph is a classical data flow graph, similar to the ones in [31, 48]. We also reuse cudf [52] ops, arrow ops [3], etc.



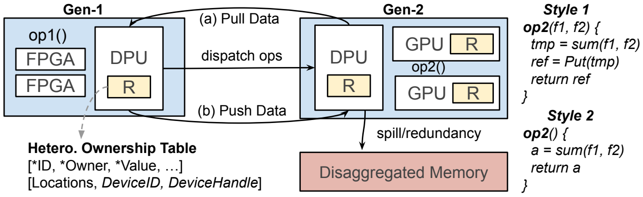
Launch Physical Graph

```
b  = [B.remote() for i in range(n)]
c  = [C.remote() for i in range(m)]
d1 = [D1-gpu.remote(b[], c[])]
d2 = [D2-fpga.remote(b[], c[])]
e  = E.remote(d1 + d2)
```

Figure 2: Skadi Architecture. *(1) The top half corresponds to the tiered access layer. (2) Vertices are ops in blue boxes. (3) The subscripts in physical graph vertices are the default parallelism degree. Dashed edges are keyed using a certain hash. (4) The bottom half is the stateful serverless runtime over a disaggregated infrastructure. If Skadi is not deployed onto tightly-coupled clusters, systems running there (e.g., [7]) can exchange data with Skadi via the caching layer. (5) The caching layer exposes KV APIs. Ideally, it can manage memory as plotted in red boxes, including host DRAM, HBM in heterogeneous devices, and disaggregated memory. The caching layer is responsible for managing data locations, replication, tiering policies etc. Users of it only see KV APIs. (6) The pseudo-code on the right shows how to launch functions in Ray to run the graph on the top.* b, c, d1, *and* d2 *are futures, Skadi has two protocols to resolve them.*

In building the access layer, we find a key challenge is defining the IR to build hardware-agnostic computations and further building FlowGraph with it. On the one hand, it should be generic enough to build computing patterns data systems commonly use. On the other hand, we should be able to lower it onto multiple hardware backends (e.g., CPU, FPGA, GPU, RMT [8]). No such IR exists today.

In response, we use MLIR [41], a compiler infrastructure for creating domain-specific compilers. Recent works [7, 14, 34] showcase that MLIR is viable to build data systems. We plan to use the open-source Daphne project [14] to build our IR because it is the closest to an ideal access layer (see Table 1): it has tiered declarative APIs, an MLIR-based DSL, and abstractions like data frames, and matrix operators. Although Daphne is versatile, its IR cannot be lowered onto distinct hardware since it only supports a single LLVM backend. In order to build hardware-agnostic IR, we must add more MLIR backends such as CIRCT for FPGA, or SPIR-V for



```
op2(f1, f2) {
  tmp = sum(f1, f2)
  ref = Put(tmp)
  return ref
}
```

```
op2() {
  a = sum(f1, f2)
  return a
}
```

Figure 3: Stateful serverless runtime details. *(1) We omit regular servers and show two physically-disaggregated devices, on top of which runs a modified raylet (the box with R). (2) We make Ray's ownership table heterogeneity-aware by adding a device ID and a handle to the device driver (DeviceID and DeviceHandle). (3) Two coding styles differ in whether the op uses opaque Ray pointers or futures not passed as parameters. Running coding style 2 requires a raylet inside heterogeneous devices.*

GPU. This is an incredibly challenging task, which remains an open research question.

A key benefit of using hardware-agnostic IR is that we can lower a single piece of code to multiple hardware backends, based on a set of predefined policies. For example, in order to compare how an op performs on two platforms, the MLIR-based vertex D in Figure 2 is lowered onto a GPU version (D1) and an FPGA version (D2) for a direct comparison. This certainly alleviates developers' chore of implementing and selecting the best hardware platform. In addition, should we finalize the degree of parallelism during the compilation time [75], or allow tuning [5] or graph reshaping [31] during runtime is an open question. We leave such policy explorations to future work.

2.3 Stateful Serverless Runtime

Skadi's stateful serverless runtime is built based on Ray. Given a physical graph description, we will launch tasks (or actors) and pass futures using Ray's vanilla distributed task and KV APIs (code snippet in Figure 2).

2.3.1 Ray Primer. Ray is a task-parallel system originally designed as a glue system to run RL workloads [46]. It has task APIs for users to launch functions using stateless tasks or stateful actors. Each Ray node runs a daemon process called raylet which is responsible for running tasks and managing a distributed object store called plasma. Functions exchange data either by value or by futures. The future refers to data residing in the distributed object store. Future resolution uses an ownership protocol [24]. Due to the limited space, we refer readers to [46, 68, 69, 81] for more details.

2.3.2 Design. We now describe how we overhaul Ray to meet our goals described in Sec §1. We first offload raylet to disaggregated devices (recall that such a device consists of a DPU along with other dominant resources like GPU or DRAM [28, 37, 45]). To access memory on heterogeneous devices from Ray's code using regular opaque pointers, we modify Ray's ownership table [69] also to include a device ID and a handle for the device communication driver. Consequently, the raylet on DPU also manages memory on its companion devices. Combined, this results in the first generation of our stateful serverless runtime, capable of managing physically-disaggregated devices and their memory (Gen-1 in Figure 3). We are building it using an in-house card that has a BlueField DPU [51] and FPGAs.

The first generation can run most use cases efficiently. But it is inefficient in running short-lived ML ops for two reasons. First, it continues to use the CPU-centric model in which the DPU orchestrates all resources of a device. The management of tasks and pointers must go through the centralized DPU. For instance, if two chained ops from the same physical graph are deployed to two different FPGAs in Figure 3, their communication (e.g., future resolution) must go through the DPU. For short-lived ML ops, frequent trips to the DPU are too costly. Second, Ray's future resolution uses a pull-based model in which the consumer pulls data from the producer on demand [69]. This creates long stalls for short-lived ops. Pathways [7] made a similar observation.

We propose a second generation (Gen-2 in Figure 3) to solve these issues. It adopts three key changes. First, we eschew the CPU-centric model by deploying a device-specific raylet to each heterogeneous device. Second, we add another push-based model for future resolution, in which the producer pushes data to the consumer proactively. Third, to resolve potential out-of-memory and to increase availability, we extend the caching layer to include disaggregated memory. This generation embraces a device-centric model and can run ops that call Ray APIs everywhere (e.g., ops written in Figure 3's style-2). We are still in the early stage of designing a CUDA-based raylet for GPU. It's intrinsically more difficult to build a similar raylet for FPGA despite its recent breakthroughs [38, 77].

3 RELATED WORK

Numerous data systems have been built over the years, e.g., parallel processing [16, 31, 75], task-parallel [35, 46, 49], large-scale ML [2, 7, 11, 79], OLAP [6, 44, 71, 72], graph processing [24, 25, 50], streaming [9, 48, 76], etc. As we mentioned earlier, cloud providers have relied on three principles to maintain cost-effective data systems. We compare these efforts in Table 1 across five dimensions.

(1) **API and IR.** We categorize the system APIs into POSIX, imperative, and declarative. Most emerging operating systems offer POSIX except FractOS [66], which exposes an imperative abstraction for users to write DAG manually. A similar approach is taken by several serverless frameworks [17, 36, 63] and task-parallel systems like CIEL [49], MODC [35], and Ray [46].

	API	IR	Serverless	PhysDisagg.	Integr.
Dist. OS [47]	POSIX	×	×	×	×
LegoOS [59]	POSIX	×	×	✓	×
FractOS [66]	I-API	×	×	✓	×
Molecule [17]	I-API	×	stateless	✓	×
Cloudburst [63]	I-API	×	stateful	×	×
Pocket [36]	I-API	×	stateful	×	×
CIEL [49]	I-API	×	stateful	×	×
Ray [46]	I-API	×	stateful	×	✓
MODC [35]	I-API	×	stateful	×	×
Pathways [7]	D-API	MLIR	stateful	×	×
OneFlow [74]	D-API	IR	actor	×	×
Dryad [31]	D-API	×	stateless	×	✓
Naiad [48]	D-API	×	statelful	×	✓
DPA [39]	D-API	×	actor	×	✓
DBOS [40, 62]	D-API	×	stateful	×	✓
TCR [20, 29]	D-API	IR	×	×	✓
DAPHNE [14]	D-API	MLIR	stateless	×	✓
Skadi	D-API	MLIR	stateful	✓	✓

Table 1: Related work comparisons. *(1) I-API: imperative APIs. D-API: declarative API. All non-POSIX systems support DAG. (2) IR: the system uses an IR for hardware-agnostic computation. (3) PhysDisagg: whether the system utilizes physically-disaggregated devices for higher cluster efficiency. (4) Integr.: whether the system runs integrated data system pipelines.*

Skadi's declarative API is inspired by early pioneers such as Dryad [31], Naiad [48], and DAPHNE [14]. Compared to DAPHNE, Skadi differs in using MLIR with multiple hardware backends to build hardware-agnostic ops [14].

(2) **Serverless Runtime.** If a system supports stateless functions, we believe it can support the serverless paradigm regardless of whether it was explicitly designed so. Dryad [31] is a prominent example. We categorize systems capable of storing function states as stateful serverless runtime. Note that we explicitly differentiate actor-based systems [39, 74], although generally they are also stateful systems. Typical stateful runtime include Cloudburst [63], Pocket [36, 64], etc. Pathways [7] has a data-flow system called Plaque which has a data store similar to Ray's plasma, so we also mark it as a stateful runtime.

(3) **Physical Disaggregation.** Not many systems are capable of managing physically-disaggregated devices. LegoOS [59] and FractOS [66] are two seminal works in this space. LegoOS logically groups physically disaggregated devices and exposes a single-system image to run unmodified applications written for monolithic kernels. FractOS eschews the CPU-centric model and allows developers to write imperative DAGs to use disaggregated devices. However, neither of them is designed for running data-intensive systems. Some work optimizes data systems running on disaggregated devices [37, 78] but mostly targets a single system. Skadi extends the Ray runtime to disaggregated devices by offloading key components to DPUs.

(4) **Data Systems Integration.** In general, systems that aim to run mixed workloads need to provide an abstraction and a runtime. In Skadi's case, we have the tiered access layer and the stateful serverless runtime, respectively. Prior works like DPA [39], DBOS [40, 62], and DAPHNE [14] also target integrated data systems. For example, DBOS can run serverless data system workloads on top of a distributed database. DPA provides an actor-based programming model for building distributed query serving systems and a shared runtime for running and scheduling actors. DAPHNE builds its abstraction layer based on MLIR and develops a runtime across heterogeneous devices from scratch.

4 CONCLUSION

Over the last two decades, cloud vendors have used three principles to keep running demanding data systems cost-effective: resource disaggregation, domain-specific computing, and serverless computing. Since each was proposed to solve a different problem, these principles fail to work harmoniously and oftentimes result in conflicts in practical deployments. This paper proposes a distributed runtime to allow them to work in concert. The distributed runtime will be the narrow waist between data systems and the datacenter infrastructure. It achieves the separation of concerns at scale: users apply domain-specific declarative computation on data, oblivious of data location, concurrency, consistency, disaggregation style, or even the hardware used to do the compute. Indeed, the envisioned distributed runtime and our implementation Skadi is an overly ambitious take. We've only scratched the surface in this paper and still have a long way to achieve our ideals.

ACKNOWLEDGEMENT

We thank anonymous reviewers for their valuable feedback. We also thank Xiaoyang Deng, Rongfeng He, Zeke Wang, and Xiaosong Lei for their insightful discussions. This work is supported in part by the National Key Research and Development Plan of China (No. 2022YFB4500400), the Strategic Priority Research Program of Chinese Academy of Sciences under grant number XDA0320000 and XDA0320300, the National Natural Science Foundation of China (Grant No. 62090022 and 62172388), and Youth Innovation Promotion Association of Chinese Academy of Sciences (2020105).

REFERENCES

[1] a16z. Emerging Architectures for Modern Data Infrastructure. https://a16z.com/2020/10/15/emerging-architectures-for-modern-data-infrastructure/.

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2016).

[3] Apache Arrow. https://arrow.apache.org/.

[4] Armbrust, M., Ghodsi, A., Xin, R., and Zaharia, M. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR* (2021).

[5] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015).

[6] Armenatzoglou, N., Basu, S., Bhanoori, N., Cai, M., Chainani, N., Chinta, K., Govindaraju, V., Green, T. J., Gupta, M., Hillig, S., Hotinger, E., Leshinksy, Y., Liang, J., McCreedy, M., Nagel, F., Pandis, I., Parchas, P., Pathak, R., Polychroniou, O., Rahman, F., Saxena, G., Soundararajan, G., Subramanian, S., and Terry, D. Amazon Redshift Re-Invented. In *Proceedings of the 2022 International Conference on Management of Data* (2022).

[7] Barham, P., Chowdhery, A., Dean, J., Ghemawat, S., Hand, S., Hurt, D., Isard, M., Lim, H., Pang, R., Roy, S., et al. Pathways: Asynchronous distributed dataflow for ML. *Proceedings of Machine Learning and Systems* (2022).

[8] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (2013).

[9] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* (2015).

[10] Caulfield, A. M., Chung, E. S., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J.-Y., Lo, D., Massengill, T., Ovtcharov, K., Papamichael, M., Woods, L., Lanka, S., Chiou, D., and Burger, D. A Cloud-Scale Acceleration Architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (2016).

[11] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (2018).

[12] Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[13] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* (2013).

[14] Damme, P., Birkenbach, M., Bitsakos, C., Boehm, M., Bonnet, P., Ciorba, F., Dokter, M., Dowgiallo, P., Eleliemy, A., Faerber, C., et al. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. In *Conference on Innovative Data Systems Research* (2022).

[15] Daoud, F., Watad, A., and Silberstein, M. GPUrdma: GPU-side library for high performance networking from GPU kernels. In *Proceedings of the 6th international Workshop on Runtime and Operating Systems for Supercomputers* (2016).

[16] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* (2008).

[17] Du, D., Liu, Q., Jiang, X., Xia, Y., Zang, B., and Chen, H. Serverless Computing on Heterogeneous Computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2022).

[18] Fowers, J., Ovtcharov, K., Papamichael, M., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., Adams, L., Ghandi, M., Heil, S., Patel, P., Sapek, A., Weisz, G., Woods, L., Lanka, S., Reinhardt, S. K., Caulfield, A. M., Chung, E. S., and Burger, D. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (2018).

[19] Frostig, R., Johnson, M. J., and Leary, C. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* (2018).

[20] Gandhi, A., Asada, Y., Fu, V., Gemawat, A., Zhang, L., Sen, R., Curino, C., Camacho-Rodríguez, J., and Interlandi, M. The tensor data platform: Towards an ai-centric database system. *arXiv preprint arXiv:2211.02753* (2022).

[21] Geyer, A., Krause, A., Habich, D., and Lehner, W. Pipeline Group Optimization on Disaggregated Systems. In *Proceedings of CIDR* (2023).

[22] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (2003).

[23] Gibson, D., Hariharan, H., Lance, E., McLaren, M., Montazeri, B., Singh, A., Wang, S., Wassel, H. M. G., Wu, Z., Yoo, S., Balasubramanian, R., Chandra, P., Cutforth, M., Cuy, P., Decotigny, D., Gautam, R., Iriza, A., Martin, M. M. K., Roy, R., Shen, Z., Tan, M., Tang, Y., Wong-Chan, M., Zbiciak, J., and Vahdat, A. Aquila: A unified, low-latency fabric for datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022).

[24] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012).

[25] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014).

[26] Google Cloud Platform. https://cloud.google.com/bigquery.

[27] Grandl, R., Singhvi, A., Viswanathan, R., and Akella, A. Whiz: Data-Driven Analytics Execution. In *18th USENIX Symposium on Networked Systems Design and Implementation* (2021).

[28] Guo, Z., Shan, Y., Luo, X., Huang, Y., and Zhang, Y. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2022).

[29] He, D., Nakandala, S. C., Banda, D., Sen, R., Saur, K., Park, K., Curino, C., Camacho-Rodríguez, J., Karanasos, K., and Interlandi, M. Query Processing on Tensor Computation Runtimes. *Proc. VLDB Endow.* (2022).

[30] Hennessy, J. L., and Patterson, D. A. A New Golden Age for Computer Architecture. *Commun. ACM* (2019).

[31] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (2007).

[32] Jouppi, N. P., Kurian, G., Li, S., Ma, P., Nagarajan, R., Nai, L., Patil, N., Subramanian, S., Swing, A., Towles, B., et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. *arXiv preprint arXiv:2304.01433* (2023).

[33] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017).

[34] Jungmair, M., Kohn, A., and Giceva, J. Designing an Open Framework for Query Optimization and Compilation. *Proc. VLDB Endow.* (2022).

[35] Keeton, K., Singhal, S., Volos, H., Zhang, Y., Chaurasiya, R. C., Crasta, C. R., George, S. T., Natarajan, K., Shome, P., Suresh, S., et al. MODC: resilience for disaggregated memory architectures using task-based programming. *arXiv preprint arXiv:2109.05329* (2021).

[36] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (2018).

[37] Korolija, D., Koutsoukos, D., Keeton, K., Taranov, K., Milojičić, D., and Alonso, G. Farview: Disaggregated memory with operator off-loading for database engines. In *Conference on Innovative Data Systems Research* (2021).

[38] Korolija, D., Roscoe, T., and Alonso, G. Do OS Abstractions Make Sense on FPGAs? In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation* (2020).

[39] Kraft, P., Kazhamiaka, F., Bailis, P., and Zaharia, M. Data-Parallel Actors: A Programming Model for Scalable Query Serving Systems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022).

[40] Kraft, P., Li, Q., Kaffes, K., Skiadopoulos, A., Kumar, D., Cho, D., Li, J., Redmond, R., Weckwerth, N., Xia, B., et al. Apiary: A DBMS-Backed Transactional Function-as-a-Service Framework. *arXiv preprint arXiv:2208.13068* (2022).

[41] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).

[42] Liu, H., Tang, B., Zhang, J., Deng, Y., Yan, X., Zheng, X., Shen, Q., Zeng, D., Mao, Z., Zhang, C., You, Z., Wang, Z., Jiang, R., Wang, F., Yiu, M. L., Li, H., Han, M., Li, Q., and Luo, Z. GHive: Accelerating Analytical Query Processing in Apache Hive via CPU-GPU Heterogeneous Computing. In *Proceedings of the 13th Symposium on Cloud Computing* (2022).

[43] Marty, M., de Kruijf, M., Adriaens, J., Alfeld, C., Bauer, S., Contavalli, C., Dalton, M., Dukkipati, N., Evans, W. C., Gribble, S., Kidd, N., Kononov, R., Kumar, G., Mauer, C., Musick, E., Olson, L., Rubow, E., Ryan, M., Springborn, K., Turner, P., Valancius, V., Wang, X., and Vahdat, A. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019).

[44] Melnik, S., Gubarev, A., Long, J. J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T., Ahmadi, H., Delorey, D., Min, S., Pasumansky, M., and Shute, J. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* (2020).

[45] Min, J., Liu, M., Chugh, T., Zhao, C., Wei, A., Doh, I. H., and Krishnamurthy, A. Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021).

[46] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., and Stoica, I. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (2018), OSDI'18.

[47] Mullender, S. J., Van Rossum, G., Tananbaum, A., Van Renesse, R., and Van Staveren, H. Amoeba: A distributed operating system for the 1990s. *Computer* (1990).

[48] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013).

[49] Murray, D. G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., and Hand, S. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (2011).

[50] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S., and Oskin, M. Latency-Tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (2015).

[51] NVIDIA. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[52] NVIDIA. GPU Accelerated Data Science with RAPIDS. https://www.nvidia.com/en-us/deep-learning-ai/software/rapids/.

[53] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library, 2019.

[54] Pemberton, N., Zabreyko, A., Ding, Z., Katz, R., and Gonzalez, J. Kernel-as-a-Service: A Serverless Interface to GPUs. *arXiv preprint arXiv:2212.08146* (2022).

[55] Pu, Q., Venkataraman, S., and Stoica, I. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (2019).

[56] Ranganathan, P., Stodolsky, D., Calow, J., Dorfman, J., Guevara, M., Smullen IV, C. W., Kuusela, A., Balasubramanian, R., Bhatia, S., Chauhan, P., Cheung, A., Chong, I. S., Dasharathi, N., Feng, J., Fosco, B., Foss, S., Gelb, B., Gwin, S. J., Hase, Y., He, D.-k., Ho, C. R., Huffman Jr., R. W., Indupalli, E., Jayaram, I., Kongetira, P., Kyaw, C. M., Laursen, A., Li, Y., Lou, F., Lucke, K. A., Maaninen, J., Macias, R., Mahony, M., Munday, D. A., Muroor, S., Penukonda, N., Perkins-Argueta, E., Persaud, D., Ramirez, A., Rautio, V.-M., Ripley, Y., Salek, A., Sekar, S., Sokolov, S. N., Springer, R., Stark, D., Tan, M., Wachsler, M. S., Walton, A. C., Wickeraad, D. A., Wijaya, A., and Wu, H. K. Warehouse-Scale Video Acceleration: Co-Design and Deployment in the Wild. In *Proceedings of the 26th ACM International*

Conference on Architectural Support for Programming Languages and Operating Systems (2021).

[57] Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N. J., Popa, R. A., Gonzalez, J. E., Stoica, I., and Patterson, D. A. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Commun. ACM* (2021).

[58] Shan, Y. *Distributing and Disaggregating Hardware Resources in Data Centers.* University of California, San Diego, 2022.

[59] Shan, Y., Huang, Y., Chen, Y., and Zhang, Y. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (2018).

[60] Sidler, D., Wang, Z., Chiosa, M., Kulkarni, A., and Alonso, G. StRoM: Smart Remote Memory. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020).

[61] Singhvi, A., Akella, A., Anderson, M., Cauble, R., Deshmukh, H., Gibson, D., Martin, M. M. K., Strominger, A., Wenisch, T. F., and Vahdat, A. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021).

[62] Skiadopoulos, A., Li, Q., Kraft, P., Kaffes, K., Hong, D., Mathew, S., Bestor, D., Cafarella, M., Gadepally, V., Graefe, G., Kepner, J., Kozyrakis, C., Kraska, T., Stonebraker, M., Suresh, L., and Zaharia, M. DBOS: A DBMS-Oriented Operating System. *Proc. VLDB Endow.* (2022).

[63] Sreekanti, V., Wu, C., Lin, X. C., Schleier-Smith, J., Gonzalez, J. E., Hellerstein, J. M., and Tumanov, A. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.* (2020).

[64] Stuedi, P., Trivedi, A., Pfefferle, J., Klimovic, A., Schuepbach, A., and Metzler, B. Unification of Temporary Storage in the Nodekernel Architecture. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (2019).

[65] Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., and Bao, X. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017).

[66] Vilanova, L., Maudlej, L., Bergman, S., Miemietz, T., Hille, M., Asmussen, N., Roitzsch, M., Härtig, H., and Silberstein, M. Slashing the Disaggregation Tax in Heterogeneous Data Centers with FractOS. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022).

[67] Vuppalapati, M., Miron, J., Agarwal, R., Truong, D., Motivala, A., and Cruanes, T. Building an Elastic Query Engine on Disaggregated Storage. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (2020).

[68] Wang, S., Liagouris, J., Nishihara, R., Moritz, P., Misra, U., Tumanov, A., and Stoica, I. Lineage Stash: Fault Tolerance off the Critical Path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019).

[69] Wang, S., Liang, E., Oakes, E., Hindman, B., Luan, F. S., Cheng, A., and Stoica, I. Ownership: A Distributed Futures System for Fine-Grained Tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation* (2021).

[70] Winter, C., Giceva, J., Neumann, T., and Kemper, A. On-Demand State Separation for Cloud Data Warehousing. *Proc. VLDB Endow.* (2022).

[71] Yandex. Clickhouse. https://clickhouse.com/.

[72] Yang, F., Tschetter, E., Léauté, X., Ray, N., Merlino, G., and Ganguli, D. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (2014).

[73] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P. K., and Currey, J. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008).

[74] Yuan, J., Li, X., Cheng, C., Liu, J., Guo, R., Cai, S., Yao, C., Yang, F., Yi, X., Wu, C., et al. Oneflow: Redesign the distributed deep learning framework from scratch. *arXiv preprint arXiv:2110.15032* (2021).

[75] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (2012).

[76] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013).

[77] Zha, Y., and Li, J. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020).

[78] Zhang, Q., Chen, X., Sankhe, S., Zheng, Z., Zhong, K., Angel, S., Chen, A., Liu, V., and Loo, B. T. Optimizing data-intensive systems in disaggregated data centers with teleport. In *Proceedings of the 2022 International Conference on Management of Data* (2022).

[79] Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., Gonzalez, J. E., and Stoica, I. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation* (2022).

[80] Zhou, Y., Wassel, H. M. G., Liu, S., Gao, J., Mickens, J., Yu, M., Kennelly, C., Turner, P., Culler, D. E., Levy, H. M., and Vahdat, A. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation* (2022).

[81] Zhuang, S., Li, Z., Zhuo, D., Wang, S., Liang, E., Nishihara, R., Moritz, P., and Stoica, I. Hoplite: Efficient and Fault-Tolerant Collective Communication for Task-Based Distributed Systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021).