Fangorn: Adaptive Execution Framework for Heterogeneous Workloads on Shared Clusters

Yingda Chen, Jiamang Wang, Yifeng Lu, Ying Han, Zhiqiang Lv, Xuebin Min, Hua Cai, Wei Zhang, Haochuan Fan, Chao Li, Tao Guan, Wei Lin, Yangqing Jia and Jingren Zhou
Alibaba Group Inc.
fangorn@list.alibaba-inc.com

ABSTRACT

Pervasive needs for data explorations at all scales have populated modern distributed platforms with workloads of different characteristics. The growing complexities and diversities have thereafter imposed distinct challenges to execute them on shared clusters in corporate or public clouds. This paper presents Fangorn, an adaptive execution framework built on an enriched graph model. As the underlying infrastructure for core computation platforms at Alibaba, Fangorn supports various execution modes and caters to heterogeneous workloads. With the capability to orchestrate graph executions with both long-running and requested-on-demand resources at the same time, Fangorn allows exploration of tradeoffs between latency and resource efficiency, for jobs of all scales. By modeling distributed job executions as mutable graphs with pluggable components, Fangorn offers a systematic framework to adjust job executions adaptively, according to data statistics collected during run-time. Fangorn supports an array of different computation engines ranging from relational to deep learning, and is fully deployed on production clusters across Alibaba. It manages tens of millions of distributed jobs daily, with job size scaling from one to half-million.

PVLDB Reference Format:
Yingda Chen, Jiamang Wang, Yifeng Lu, Ying Han, Zhiqiang Lv, Xuebin Min, Hua Cai, Wei Zhang, Haochuan Fan, Chao Li, Tao Guan, Wei Lin, Yangqing Jia and Jingren Zhou. Fangorn: Adaptive Execution Framework for Heterogeneous Workloads on Shared Clusters. PVLDB, 14(12): 2972 - 2985, 2021.
doi:10.14778/3476311.3476376

1 INTRODUCTION

Decades into the development of big data stacks, data-parallel computations today encompass a much wider spectrum of applications than before. The diversities of distributed workloads manifest themselves, not only in the various computation patterns (e.g., relational vs machine learning), but also in the vastly different amount of data processed per job and different SLA expectations. At one end of the spectrum are *batch* workloads challenged with processing petabytes of data, whose scales and complexities continue to grow. At the other end, are *interactive* data analytics tasked with time-sensitive data exploration, where latency is critical to support timely decision making. Characteristics associated with various workloads entail distinct, sometimes conflicting, requirements on scheduling and execution. For example, batch processing [2, 8, 11] usually follows Bulk Synchronous Parallelism(BSP) [39] model, and place scheduling barriers between data-flow stages, to ensure efficient resource usage and data-recovery in presence of failures. In contrast, continuous data pipelining offers execution acceleration prioritized by *time-sensitive* interactive engines [4, 15, 38], which often leverage gang-scheduling to fully enable data pipelining, and to avoid process-launching overhead for various job components. In addition, modern data processing such as deep learning [1, 24] reassembles computation patterns that may not be readily formulated by acyclic data-flow [10, 14, 36]. The diverging requirements on execution frameworks from different workloads, some as fundamental as scheduling granularities, resource types and lifespans, have led to silo-solutions that host data processing on different frameworks, based on input sizes and/or computation patterns. Many such purposefully-built engines are developed with proprietary execution frameworks built-in, tailored to targeted scenarios. Such choices, however, not only introduce engineering overhead associated with duplicate development and maintenance, but may also fall-short for workloads with scales and/or characteristics that fall *between* sweet-spots targeted by different frameworks.

This paper presents Fangorn, the core execution framework at Alibaba that supports the diverse workloads across the company, and for its enterprise customers on public cloud services. Fangorn is deployed on production clusters with a total of over 100,000 physical machines. More than *15 million* distributed jobs are orchestrated daily to process *Exabytes of data*, with job scales ranging from 1 to over *half-million*. It empowers a state-of-art proprietary big-data platform MaxCompute [27] with multitudinous workloads that span from agile interactive analytics, to massively-parallel batch processing. In addition, it underpins Alibaba's machine learning and deep learning platform [26] that hosts multiple distributed learning engines such as TensorFlow and PyTorch [1, 24]. Other workloads on Fangorn come from distributed graph neural network [49], automatic data placement[12], and many others. The design and implementation of Fangorn are motivated by observations from production, to support the heterogeneous workloads on multi-tenant clusters. For example, complex computations at massive scale present common challenges against composing optimal execution plan beforehand, urging execution framework to be able to adapt dynamically during job runtime. Additionally, jobs composed by different computation engines may not fit into one prescribed execution model, a versatile framework that accommodates various computation patterns is therefore of vital importance.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476376

Several notable characteristics distinguish Fangorn from previous works. *Firstly*, Fangorn is built upon a new graph model capable of describing various workloads. Inspired by earlier works [3, 14, 36], Fangorn models execution of a distributed job as a *graph*. Unlike the directed acyclic graph (DAG) modeling where edges represent data channels and execution is driven by data-flow [14], Fangorn annotates edges and stages with enriched physical properties. It provides native support for modern computation patterns in addition to legacy batch processing. For example, *concurrent* connection inherently supported on Fangorn provides essential descriptions for deep-learning workloads based on parameter server [20], in addition to providing a unified model for co-scheduling nodes to enable data-pipelining. *Secondly*, Fangorn is adaptive. It extends beyond canonical dynamic strategies such as parallelism adjustment based on partition-collapsing. Instead, late bindings of both physical and logical graphs are enabled on Fangorn. It facilitates real-time adjustment of physical graph properties, and reconfiguration of logical execution plan (e.g., *lazy* join algorithm selection). *Finally*, Fangorn is hybrid in nature. The framework manages a pool of long-running containers, in supplement to those spontaneously-requested from resource manager. It can orchestrate jobs individually by a dedicated job manager, or via a long running execution *service*. Fangorn's underlying graph model allows it to harness various resources *within* one single job. This facilitates exploration of tradeoffs suitable for workloads of all scales. Fangorn is platform-agnostic and can orchestrate jobs on Fuxi [47], YARN [40] and Kubernetes [30].

We summarize key contributions of this paper as follows:

- We present Fangorn as an adaptive execution framework built on a descriptive graph model that extends beyond canonical acyclic data-flow, with native support for workloads of different types, scales, and characteristics.
- We generalize dynamic graph reconfigurations to late binding of both physical and logical graphs during execution, enabling new paradigms of adaptive executions that address a broader class of practical challenges on distributed workloads. More importantly, they are resolved *systematically* within the same unified framework, without manual interventions for each individual scenario.
- We present a hybrid architecture capable of leveraging both long-running, and spontaneously-requested containers, within one job. Explorations are made among various tradeoffs, for workloads of all scales that prioritize different system metrics and business SLAs. The architecture additionally enables formulation of special scheduling strategies for heterogeneous hardware (CPU, GPU, FPGA etc.).
- We evaluate performance by considering latency improvement and resource-efficiency jointly, against standard benchmarks and production workloads on enterprise platforms, revealing how tradeoffs can be made on multi-tenant clusters where resources are not abundantly available, and how adaptive executions help address realistic production issues.

The rest of paper is structured as follows. Section 2 describes characteristics of production workloads, which motivate the Fangorn architecture presented in Section 3. Section 4 is dedicated to describing how graphs can adapt to runtime statistics and cluster uncertainties on Fangorn, for optimal executions. Resource-aware hybrid executions are then discussed in Section 5. Finally, performance over standard benchmarks and production workloads are evaluated in Section 6, before we conclude the paper in Section 7.

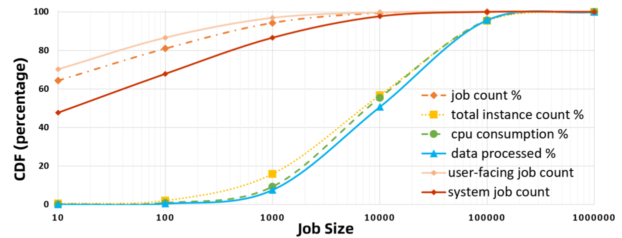

Figure 1: Characteristics of production workloads

2 WORKLOADS CHARACTERISTICS

In this section, we first set the stage by presenting important observations made from production workloads, and key challenges faced by previous frameworks before Fangorn is developed. While some observations may be attributed to unique characteristics of workloads at Alibaba, we believe many of them, and the associated challenges, resonate with technology companies hosting large-scale multi-tenant clusters and/or providing cloud services [5, 41].

a) Job Scales and Categories: Fig. 1 profiles several important characteristics of production workloads. Of the tens of millions job executed per day, a remarkable fraction are of relatively small scales: jobs of size 100 or less account for 80% of total job count. However, for resource-related metrics, be it physical instance count, resource consumption, or the amount of data processed, medium-to-large workloads with scale between 1000 to $100,000$ are clearly dominant. Many of these heavy-lifting workloads assist business-critical decisions by aggregating large volume of data, while smaller workloads facilitate interactive processing and agile business analytics. Additionally, jobs from system pipelines tend to be executed in larger scales, still a notable fraction of them are not necessarily massively-parallel. Serving the heterogeneous workloads requires the underlying execution framework to be capable of hosting a huge amount of interactive workloads, while tackling challenges imposed by large-scale jobs. The unification is not only desirable, but oftentimes *imperative*, towards productive data processing systems. In terms of workloads categories, relational queries dominate by job count: accounting for over 90% of total. Meanwhile, non-relational workloads such as machine learning/deep learning are often more compute-intensive and account for a very notable portion of cluster resource consumption, despite their relatively small number. They also consume *majority* of GPU resources in production clusters.

b) Resource Contention: Fig. 2 profiles CPU usage on a typical production cluster (with several thousand physical machines) from 21:00pm to 6:00am. We can see that, scheduled resource usage quickly saturates and stays at around 100% after midnight, with real CPU usage fluctuating around 80%[1]. This usage pattern is quite typical for large enterprise clusters, since once a full day's data is available after mid-night, large-scale batch workloads are launched to derive business-critical insights, with stringent SLAs

[1]Note that due to co-existence of non-computational workloads on production cluster, aggregated CPU usage in Fig. 2 spikes up *before* midnight, while the peak of computation workloads may lag a bit.

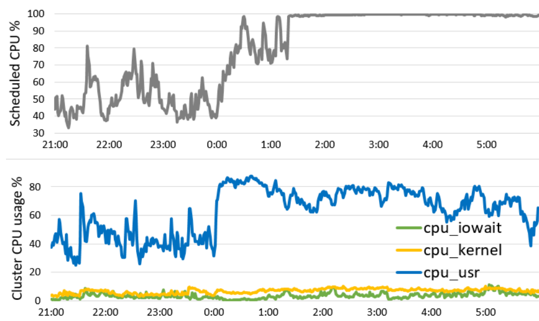

Figure 2: Characteristics of cluster resource usage

for reports to be available next morning. During peak hours, requests queue up and anything released is immediately re-assigned. Therefore, resource usage efficiency is extremely critical. Generally, batch executions that request resource spontaneously are known to be resource-efficient. However, the simple strategy of queuing up things of all scales runs the risk of starving smaller workloads. More often than not, meaningful business report relies on the output from multiple distributed jobs of various scales, with complex inter-job dependency. Determination of optimal workflow coordination is a challenging problem in itself, however, it is unquestionable that ability to balance resource usage and execution latency is of paramount importance in designing underlying execution framework. Finally, modern data processing, such as deep learning, depends on specialized hardwares with distinctly different characteristics, which shall be addressed by execution framework as well.

c) Sub-optimal Execution Plans are just facts of life associated with distributed workloads, no matter how sophisticated the application optimizer is implemented. This is particularly true in production where necessary statistics can be missing or incomplete, data manipulation may be too complex for optimizer to reason about, or sometimes completely imperceptible with user defined computation logic (for example, UDF are found in over 20% of production jobs). These are common challenges faced by various distributed engines from relational to machine learning, and can lead to sub-optimal plans that result in prolonged executions, inefficient resource usage, or even failures. Meanwhile, the option to carefully fine-tune all workloads quickly becomes infeasible with overwhelmingly large job number. Thus it is particularly important for underlying execution framework to be able to take on a potentially sub-optimal initial plan, and adapt dynamically throughout execution lifecycle, as real-time data statistics become available.

3 THE FANGORN FRAMEWORK

To support the multitude of distributed applications, and the different computation patterns entailed, Fangorn embraces a hybrid architecture capable of both individual job management and centralized multi-job services. Such hybrid framework is unified by an underlying graph model capable of encapsulating the disparate properties necessary to describe various execution characteristics accurately. Each Fangorn ***job*** is modeled by a ***graph***, composing of multiple ***stages*** connected by directed ***edges***. Edges in Fangorn graph are not necessarily bound to data-flow, rather, it encapsulates both scheduling order and data transportation (if any). A stage can be materialized into multiple parallel ***tasks*** that jointly carry out

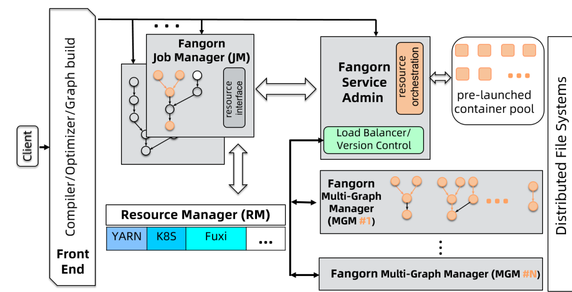

Figure 3: Overall architecture of Fangorn

the computation assigned. Each task is executed within a ***container***, which represents a slice of resource on physical machine. Heterogeneous resources are found on production clusters, referring to both resources of different life-spans (e.g., short-lived resource requested on-demand vs long-running ones) and resources associated with different hardware types (e.g., CPU vs GPU).

3.1 Architectural Overview

The overall Fangorn architecture is depicted in Fig. 3. Particularly, Fangorn framework consists of multiple components including Job Manager(JM), Fangorn Service Admin(or simply Admin), Multi-Graph Manager(MGM), and a pre-launched container pool[2] managed by Admin. Client submissions first go through frontend, where compilation and/or optimization[3] take place. Execution graphs are built with Fangorn API before submitting for executions. Fangorn exposed a rich set of graph-building APIs to facilitate accurate descriptions of various workloads that may or may not be data-flow driven. Frontend service leverages the APIs to build execution graph that serves as input to Fangorn framework. At submission, physical properties that materialize execution plan may be left vacant, such as stage parallelism; or specified as a place-holder subject to runtime adjustment, such as edge shuffle pattern. The various components in Fangorn framework all share the same graph model, with unified graph semantics and state-machine implementation. Such is the foundation for all Fangorn components to interact, and to facilitate various execution modes and strategies.

A Fangorn JM manages execution of *one single graph*. With its own dedicated resource quota, JM can accommodate computation associated with sophisticated dynamic graph adjustment, and finer-grained check-pointing to enable incremental failover recovery. As such, JM is capable of managing more complex graph, and at larger scales. Fangorn's resource interface abstracts the interactions between JM and cluster Resource Manager(RM). Additionally, it also facilitates interactions with Fangorn Service Admin, allowing resources to be requested from container pool.

Complementary to the *one-manager-per-job* model with JM, Fangorn also offers execution acceleration for time-sensitive workloads via dedicated Fangorn service. It consists of the service Admin that manages a pool of pre-launched containers, and a number of MGMs. As its name indicates, a MGM is capable of orchestrating multiple graphs, simultaneously. This includes negotiating resources with

[2]The pre-launched containers host long-running processes ready to accept new workloads, and only perform context-cleaning at the end of computation without exiting.
[3]Workloads could differ slightly in submission or execution process. For example, TensorFlow script compilation may be deferred until execution.

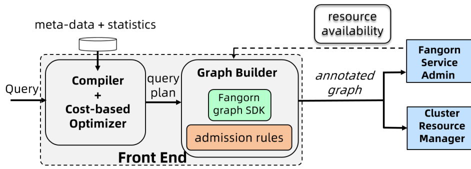

Figure 4: Job submission for relational workload

Admin, managing graph state-machines, and interacting with external system components such as storage service. On the other hand, the Admin manages the pre-launched container pool, and is responsible for delegating job submissions to one of Fangorn MGMs for execution. Both Admin and MGMs are long-running processes. Together they provide job orchestration service that avoids overhead of launching one JM per job. The number of MGMs in a Fangorn service can automatically adapt to cluster dynamics such as QPS, size of container-pool, and complexity of graphs being executed, as discussed in 3.4. It should also be noted that the pre-launched containers in the pool are not reserved exclusively for interactive workloads managed by MGMs. Instead, since different scheduling components on Fangorn are all built upon a unified graph model, JM may also request resource from this same pool to execute partial graph segments, for *hybrid executions*. Conversely, MGM could request resource from cluster RM too, which is less usual with MGM though: since it typically targets interactive workloads and it makes less sense to go through queuing and dispatching overhead common at RM. For the rest of paper, we refer hybrid executions mainly to the setup where a JM requests hybrid resources from both RM and Admin for graph execution.

Finally, while Fangorn interacts closely with various computation engines to enable adaptive executions and to explore optimal plans, components of these engines are *not considered part of Fangorn execution framework*. Therefore although technical background is provided when necessary, design and specifics of these components, such as cost-based relational optimizer, or deep-learning compiler, are beyond the scope of our discussions in this paper.

3.2 Job Submission

The decision to execute a graph via a dedicated JM or on the long-running Fangorn service, is made jointly by Frontend and Fangorn. Fig. 4 illustrates job submission flow for relational workloads: a query is firstly compiled into query plan, graph builder then iterates through the plan, and transforms it into annotated Fangorn graph using Fangorn's graph-building APIs. This graph plan carries descriptions for stages and edges to be interpreted and dynamically adjusted(when applicable) by Fangorn execution framework. This decision whether the graph shall be submitted to Admin or cluster RM (to launch dedicated JM), is made by weighting on several factors, including a) expected job size, b) real-time resource availability in the container pool, c) graph admissibility to Admin, including expected execution time, job priority, security requirements, etc., some of which are discussed in Section 5.

3.3 Graph Execution Models

To unify heterogeneous workloads, Fangorn is built upon a graph model that provides accurate descriptions for various combinations of both data-flow and execution-flow. While taking inspirations

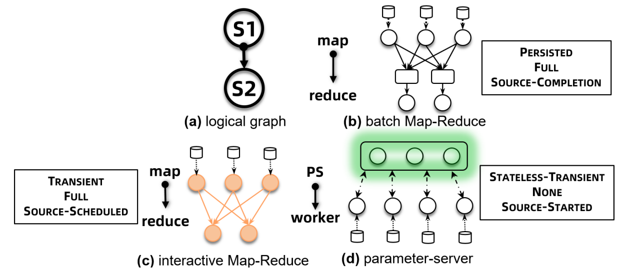

Figure 5: Physical materializations of logical graph

from earlier works [14, 36] that modeled distributed executions as DAG, Fangorn makes major extensions in graph modeling to decouple edge connections from acyclic data-flow. In addition, generalization of edges and stages in Fangorn facilitates embodiment of a logical graph into distinct physical materializations.

Table 1: Physical Edge Properties in Fangorn.

Physical Edge Property	Annotation Values
Data-Transportation	PERSISTED, TRANSIENT, STATELESS-TRANSIENT, BUFFERED
Shuffle-Pattern	NONE, FULL, CUSTOMIZED
Scheduling-Trigger	SOURCE-COMPLETION, SOURCE-SCHEDULED, SOURCE-STARTED, SOURCE-PROGRESS, MIXED

3.3.1 Edge and Stage Properties. Table 1 lists physical edge properties that can be used to annotate logical edges in Fangorn, where:
a) Data-Transportation describes the nature of physical data transfer, including physical medium involved, across an annotated edge.
b) Shuffle Pattern describes routing of data across different tasks between stages. The canonical all-to-all shuffle is described by FULL pattern, while CUSTOMIZED facilitates plugins to customize data routing. Fangorn also allows a NONE pattern to describe either absence of data flow, or data exchange occurring "out of band". As the execution infrastructure, Fangorn is responsible for ensuring that shuffle data is routed as intended. Engine runtime of each individual task simply assumes that raw-bytes is ready upon launching and can deserialize data according to its own specifications.
c) Scheduling-Trigger describes scheduling semantics. It extends beyond conventional data-flow-driven scheduling, which was built on assumption that launching of downstream task(s) is driven by production of upstream output. Instead, stages connected by an edge can execute sequentially or concurrently, with scheduling triggered by non-data-driven event. This abstraction facilitates unification of scheduling suitable for various workloads, including batch, gang-scheduling, machine learning and many others.

On the other hand, stages in Fangorn are associated with physical properties such as resource specifications and parallelism. Together, they bind a logical graph to its physical presentation. Fig. 5 exemplifies how a simple logical graph shown in Fig. 5(a) can be materialized into various physical execution graphs corresponding to different computation patterns. The canonical batch map-reduce in

Fig. 5(b) is connected by an edge with *triplet* annotation {PERSISTED, FULL, SOURCE-COMPLETION}. It denotes that full-shuffled intermediate data is persisted on disk between stages, and is fully recoverable. In terms of scheduling, SOURCE-COMPLETION indicates that downstream stage is scheduled *after* completion of upstream tasks. In comparison, interactive map-reduce is described by Fig. 5(c), with {TRANSIENT, FULL, SOURCE-SCHEDULED} triplet. The TRANSIENT implies intermediate data is only consumable as being produced (e.g., pipelined via network). In addition, it carries subtle implication that, failure in any of the tasks up- or downstream will lead to rerun of both stages. SOURCE-SCHEDULED indicates that downstream stage is jointly-scheduled with its upstream source. Notably, gang scheduling can be described by a graph whose edges are *all* annotated with SOURCE-SCHEDULED. Finally, Fig. 5(d) depicts parameter server [20](PS) model with {STATELESS-TRANSIENT, NONE, SOURCE-STARTED} triplet. One most distinct scheduling requirement for such workload is that Workers must be running concurrently with PS, and cannot progress meaningfully before all PS tasks are running. Such dependency is accurately described on Fangorn with SOURCE-STARTED scheduling trigger. During execution, data exchanged between PS and Worker contains mostly parameters not managed by the framework. Fangorn models such exchange as "out of band" by NONE shuffle. In addition, since data exchanges only occur when both PS and Worker are running concurrently, it is transient in nature. Yet TRANSIENT transportation implies that a faulty downstream will lead to upstream failure, while PS is in fact self-sustainable against Worker failures. Therefore it is more accurately annotated as STATELESS-TRANSIENT. The scheduling strategies conveyed via the triplet corresponds to subtle yet important behaviors that define characteristics of deep learning workloads.

3.3.2 Fault Tolerance. Fangorn's graph modeling also encapsulates various fault-tolerance strategies via physical edge and stage properties. For example, fault-tolerance granularity can be inferred from "Data Transportation" property of a stage's connecting edge(s). Task-level fault tolerance can be achieved for a stage whose connecting edges are all annotated with PERSISTED, since any failed task can be rescheduled individually, by recovering from persisted intermediate data. On the other hand, any failure within a stage connected to TRANSIENT edge(will lead to rerun of all connected stages: the failure radius extends until recoverable data is reached. Additionally, Fangorn infers failover strategies from Shuffle-Pattern and Scheduling-Trigger too: failure in a task shall only impact downstream tasks that have been scheduled, and with specific data dependency.

3.3.3 Execution Modes. Fangorn's descriptive graph provides the infrastructure to unify various execution strategies that explore both long-running and spontaneously-requested containers. Fig. 6 exemplifies typical execution modes for relational workloads on Fangorn. Fig. 6(a) describes batch workloads that persist all intermediate data before terminating any task, and only requests resources after all upstream tasks complete. By doing so, it achieves high reliability with efficient resource usage, making it especially suitable for massively-parallel data processing. In contrast, time-sensitive interactive workloads can opt for accelerating execution whenever possible. As such, they are gang-scheduled with pre-launched containers, and leverage data pipelining to expedite processing, as

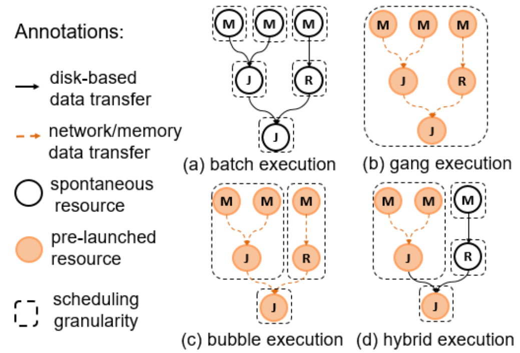

Figure 6: Graph execution modes on Fangorn

shown by Fig. 6(b). Such execution strategy entails rerun of entire graph though, upon single task failure, which can be costly. Batch execution and gang scheduling sit at the two extremes of the spectrum for distributed data processing. However, many workloads can fall *between* these two extremes. Fig. 6(c) and (d) illustrates two alternative execution modes enabled on Fangorn, namely *bubble execution* and *hybrid execution*, that explore tradeoffs between the extremes, to seek balance among execution latency, resource efficiency and fault tolerance. Bubble execution relaxes job-level gang scheduling, by dividing a graph into multiple "bubbles", each scheduled as (sub)gangs executed on pre-launched containers. In comparison, hybrid execution leverages Fangorn's unique architecture and further relaxes the exclusiveness on resource type, thus allowing co-existence of hybrid resources in the same graph. The flexibility provided by hybrid execution facilitates its wide adoption on workloads of various scales, and is discussed in Section 5.

3.4 Execution Framework At Scale

Serving multi-million jobs per day demands an execution framework that scales with increasing job count. While JM faces its own challenges of orchestrating massive-scale job adaptively and responsively, it scales naturally in terms of increasing job count: since one JM is allocated per job. In contrast, the Fangorn Service is tasked with orchestrating all admissible workloads on multi-tenant clusters, therefore it must be able to scale horizontally with increasing concurrent-job-count in the system and graph complexities.

For a job orchestration service, binding resource administration and job management into one single process simplifies overall system design, and is adopted by distributed MPP platform [38]. However, it creates a single-point system bottle-neck, a vulnerability that becomes more pronounced when deployed over large clusters with *dense* computation capabilities. *The decoupling of resource management by Admin*, and *job management by MGMs*, is a conscious design choice by Fangorn in recognition of such challenges, to offload the relatively compute-intensive duties of job management to multiple MGMs. Meanwhile, the dispatching of jobs is accomplished by load-balancing at Admin, according to real-time loads at each MGM. The service is designed to scale up or down *automatically*, in response to real-time service workloads. Particularly, the Admin will spin up new MGM(s) in presence of overwhelming influx of job submissions, or shut down existing ones gradually when service load lowers. The job delegation by Admin also provides means to resolve engineering challenges with

version control and backward compatibility. As multiple MGMs can be configured to manage jobs intended for different engineering releases, it offers mechanisms to facilitate migrations among upgrades, even in the presence of breaking changes.

On the other hand, container-administration at Fangorn Admin allows accurate resource management and optimal allocation strategies to be achieved. Comparing to alternative approach that partition resource pool vertically into multiple *sub-pools* each used exclusively for part of job submissions [4], the solution adopted by Fangorn avoids resource fragmentation, and preserves the framework's capability to offer seamless upgrade between releases. In addition, such architecture allows Admin to optimize task placement globally, to avoid unnecessary IO and to accelerate graph execution further. The responsiveness of Admin is achieved via a lightweight event-driven implementation similar to Actor model [22], with minimum blocking operations on scheduling decision-makings.

4 ADAPTIVE GRAPH EXECUTION

Crafting *the* optimal execution plan beforehand is a notoriously hard problem [18, 19], even more so for production workloads. First of all, qualities of plans rely heavily on accuracy of *compile-time statistics*, which may be incomplete or even missing for data ingested via various inlets. Secondly, as data goes through complex transformations throughout the graph, its characteristics can change dramatically, obsoleting optimality of pre-composed plans. The uncertainties can be further amplified in presence of user-defined logic, which are mostly black-box to optimizer. Moreover, for modern workloads such as deep learning, characteristics of specialized hardware can impose additional challenges against accurate planning before submission. The capability for execution framework to adapt, *throughout lifecycle of a distributed graph*, is vital to offset negative impacts from sub-optimal plans. With a fine statistics collection framework, Fangorn incorporates run-time information on intermediate data, such as record count, distribution statistics, and operator metrics, to maintain up-to-date profiling of job execution. This allows Fangorn to perform not only canonical physical graph adjustment, such as dynamic parallelism [14, 32, 36], but also to explore a much wider spectrum of dynamic reconfigurations on both physical and logical execution graphs. In this section, we discuss dynamic graph adaptations in both categories, via concrete examples implemented across multiple distributed engines.

4.1 Adaptive Handling of Data Skew

Skewness is arguably the most common cause for prolonged execution of distributed workloads, and can be mainly attributed to two factors: execution skew caused by by *stragglers* [9] and data-skew introduced by imbalance of data distribution. The former usually results from hardware anomalies or resource contentions and can be mitigated via *speculative execution* [36, 46]. Skewness in data, on the other hand, is inherent within the workloads and can get amplified throughout the graph, making it more challenging to pin-down before execution. In this sub-section, we focus mainly on how Fangorn detects and handles *data skew* automatically. Ever since MapReduce era, data skew and techniques for its mitigation have been studied extensively [13, 16, 17]. Some are built on tuning that requires domain knowledge and user intervention, which is hard to automate in production systems. Others rely on cost models derived from accurate source data statistics, which unfortunately may not always be available in production. In addition, for graph-based executions that extend the legacy two-stage MapReduce paradigm, quality of cost models can deteriorate quickly as execution graph becomes *deeper* and data goes through more rounds of complex transformations. To those ends, we believe that adaptivness in underlying execution framework is essential for skew-mitigation to be practical in production, and to be systematically-applicable.

4.1.1 Skewness with Dynamic Partition Insert. To start with, consider *dynamic partition insertion* (DPI) that persists data according to partition-keys determined during execution. DPI is commonly-used in batch ETL workloads, a simple example can be given as:

```
INSERT OVERWRITE
TABLE partitioned_sales PARTITION(country)
SELECT item, price, country FROM sales;
```

It is important that for output from DPI, data from the same partition be co-located when persisting to disk. Otherwise, output fragmentation can create overwhelmingly-large number of files, in the order of input parallelism *times* cardinality of partition keys. This large number of (often small) files can be detrimental to underlying distributed file system. Appending a reshuffle stage mitigates such risk. When output is *evenly distributed*, vanilla map-reduce paradigm handles the problem efficiently, as illustrated by Fig. 7(a). However, when data skews towards particular key(s), sizes of partitions can vary dramatically, creating imbalance among data assigned to each reduce task and leads to prolonged executions, as exemplified by Fig. 7(b). With adequate statistics, query optimizer may be able to mitigate such skew with plan that offers fine-grained repartitioning [48]. Th optimizer-oriented approach, however, relies on accurate histograms on data cardinality against specific partition keys. In general, this information is rarely readily-available in production. Furthermore, for realistic workloads, DPI is usually embedded in a complex query, and partitioned insertion only occurs at the end of a *deep* graph. This hinders practicality of cost-based solutions, since predicting accurate partitioning distribution far out from source input can be very challenging.

In comparison, late-binding on Fangorn allows the partitioning decision to be deferred until necessary data statistics have been collected. By estimating cardinality associated with valid partition keys, JM can infer optimal data partitioning strategies to ensure fair amount of shuffle data be assigned to each downstream task, resulting in the "adaptive shuffle" illustrated in Fig. 7(c). In this example, partition #0 and #1 are grouped to be processed by one single task, while the skewed partition #2 is split into multiple segments, and assigning to different tasks. Particularly for DPI, close-to-uniform output distribution can result, even in the presence of *single-key skew*. This is achievable since shuffle partitioning can be relaxed in DPI, to the extent that output from the same partition can be safely divided and distributed to different downstream tasks. Such property is not universal and Fangorn depends on query optimizer to annotate plan with such property. By leveraging Fangorn's dynamic infrastructure, adaptive shuffle have been observed to resolve skewness in range of tens to several-thousand folds in production, without any manual interventions from end-users.

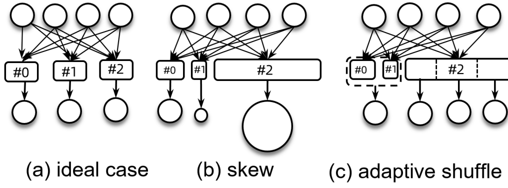

Figure 7: Skew mitigation for dynamic partition insert

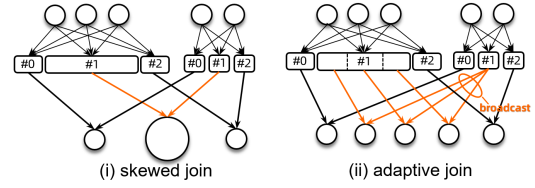

Figure 8: Adaptive Join against join skewness

4.1.2 Adaptive Skew-Join. Data skewness associated with joins has attracted extensive attention distributed relational data-processing community [7, 21, 43, 44]. In particular, a taxonomy on exploring graph topology to mitigate skew-joins is laid out in [7]. The embodiment of these studies, however, requires specific solutions to identify skew before determining optimal execution plans. In earlier studies, such plans are usually *pre-composed* by additional aggregation jobs [7], to collect necessary statistics. In contrast, Fangorn offers the infrastructure to automate run-time skew detection and dynamic plan adjustment, which is crucial for productionalizing advanced join skew mitigation techniques.

Fig. 8 demonstrates how adaptive skew join is implemented on Fangorn, which leverages B-SkewJoin [7] strategy (also known as Partial Replication Partial Redistribution [44]) in a *fully-automated* way. As statistics are collected and aggregated from multiple output of join-sources during execution, Fangorn can infer potential skewness in partitioned data. Once do, graph topology and shuffle patterns adjustment will be triggered, to spread-out skewness across multiple join tasks. In the two-way join example of Fig. 8, the (left) skewed partition is split up and assigned to 3 parallel join tasks, so that input size of each task is below a configurable threshold. In the meantime, the corresponding non-skewed right partition is *broadcast* to all 3 tasks, ensuring pair-wise joins to be correctly performed. The implementation of adaptive skew join involves not only dynamic decision-making during execution, but also adjustment of data transportation strategies among physical edge connections. This is enabled by Fangorn's *pluggable* edge descriptor, which can be tailored and dynamically adjusted to encapsulate appropriate Customized shuffle patterns. As such, Fangorn can cater to a wide class of join skew problems, including skews that occur at different partitions and from multiple join inputs. In addition, alternative strategies to mitigate join skews can be implemented as Fangorn plugins. We evaluate performance for adaptive skew join later in Section 6, against large-scale benchmark.

4.2 Conditional and Control Stages

Delaying some of the decision makings for plan optimization to run-time and allowing execution framework to reconfigure plan as needed, provides an effective means to rectify plans that may otherwise be sub-optimal. While the methodology is well-recognized, hitherto previous systems [14, 36] mostly consider adjusting graph's *physical* properties via late-binding, while the original logical graph was deemed *immutable* after submission. Nevertheless, scenarios that necessitate late-binding of *logical* graph are common for production workloads as well. To this end, continuous optimization is proposed [6] to explore dynamic interactions between query optimizer and execution framework, such that a new query plan can be recompiled if needed. Alternatively, Fangorn adopts conditional execution to address plan-uncertainties. It allows *deferrals* of final plan binding by *incorporating multiple (sub)plan-candidates at job submission*, which will be reconciled into the final plan during execution. Particularly, when optimizer cannot determine *the* optimal plan P^* among possible candidate (sub)plans set $\mathcal{A}_P$, it can construct a *sub-plan selection function* $f : \mathbb{V} \mapsto \mathcal{A}_P$, where $\mathbb{V}$ describes the entire space that relevant runtime statistics can take value from. For $\mathbf{X} \in \mathbb{V}$, f describes the mapping with guarantee:

$$f(\mathbf{x}, \mathbf{c}) = P^* \ : \ C(P^*|\mathbf{X} = \mathbf{x}) \leq C(P|\mathbf{X} = \mathbf{x}), \forall P \in \mathcal{A}_P,$$

where $\mathbf{c}$ describes additional constant parameters, and $C(\cdot)$ denotes cost function corresponding to a given plan, both supplied by optimizer. Once statistics necessary to finalize the cost function is collected during execution, the optimal plan can be determined.

4.2.1 Conditional Join. Consider, for example, a query that correlates user's monetary `spend`, with shopping session `duration`, by joining `orders` and `activities`, where `orders` is a terabyte table and `activities` measures dozens of gigabytes:

```
SELECT a.spend, a.id, b.duration FROM (
SELECT spend,id FROM orders WHERE spend > 100)a
JOIN ( SELECT userid,duration FROM activities
WHERE  duration > 60)b
ON     a.userid = b.userid;
```

Even for such a simple query, optimizer may face the dilemma of join-algorithm selection. Primarily, *selectivity* of `duration > 60` predicate determines whether broadcast hash join shall be used over the canonical sort-merge join. With low selectivity, output would be small enough (say, less than 100MB) to fit into memory assigned to a single-task, thereby facilitating broadcast hash join that performs join via in-memory hash-table lookup. It erases the necessity to sort and shuffle entire `orders` table, and avoids any side-effects associated with shuffle operations (such as shuffle skew), which makes it a more efficient distributed join algorithm, *when applicable*. However, were the selectivity higher, output from the predicate will not fit into a single task's memory, causing out-of-memory(OOM) failure, or frequent memory-disk swapping that significantly slows-down execution. When faced with the difficult decision of choosing among plan candidates whose optimality depends on varying intermediate output, optimizer tends to be more "conservative", especially when aggressive optimization bears unwelcoming consequences of job failure that can break SLAs in production. Opting for conservative plans, however, can lead to performance degradation associated with sub-optimal choice.

Fangorn resolves such dilemmas with introduction of special graph stages, namely *control* and *conditional* stage, to allow *lazy join-algorithm selection* using conditional join. A control stage is

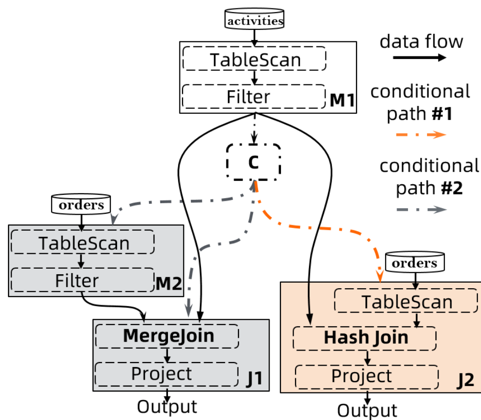

Figure 9: Conditional join example

inserted into a graph to make *control decisions* that dynamically adjust downstream graph topology and execution. A control stage can remain as a logical entity without being materialized into physical task(s), or otherwise. Conditional stage is one that may be subject to cancellation during execution. They are usually *planned with duplication*, with the expectation that only one set of the duplicated stages will be executed as selected, often by a control stage.

For the query discussed above, the resulting *conditional* join plan is shown in Fig. 9, where shaded *M2, J1, J2* are conditional stages that, depending on out characteristics of *M1*, may or may not be executed. Specifically, $\mathcal{A}_P = \{\langle M2, J1\rangle, \langle J2\rangle\}$, $\mathbf{X}$ denotes cardinality of *M1*'s output, and $\mathbf{c}$ refers to *threshold* value that indicates whether it may fit into single task memory. In this example, *C0* is merely a *logical control entity*, which collects runtime statistics (output cardinality in this case) and decides on which set of downstream conditional stages shall be executed. When $\mathbf{X} \leq \mathbf{c}$, hash join is selected with conditional path $\langle J2\rangle$, otherwise $\langle M2, J1\rangle$ is chosen to perform merge join. Non-chosen conditional stage(s) will be cancelled by Fangorn and removed from the execution graph.

Conditional join now applies to more than 40% of production relational workloads with join operations, covering simple workloads such as the one illustrated in Fig. 9, as well as complex cases where execution plan may not be finalized until multiple nested decisions have been made. The ability to steer around occasional bad cases allows optimizer to be less conservative, to explore more optimized plans: 98% of conditional joins in production favor the efficient hash joins, with OOM failure reducing to *practically none*.

4.2.2 Late Resource and Strategy Binding. Most deep learning (DL) engines offer a wide range of execution options to *end users*, such as parallelism, resource specification, and distribution strategy. While such flexibility allows expert-level users to more actively engage in execution tuning, it also imposes challenges on average users to configure execution parameters properly. With the uncertainties associated with reasoning about optimal configurations for DL workloads, we observe an apparent tendency for end-users to *over-configure*, just to be "safe". Take TensorFlow for example, the `ParameterServerStrategy` distribute strategy [28] is almost always chosen for distributed learning regardless of model size. Over-requesting resources is quite common too. For example, a user often requests 8×1 GPU while in reality the workload only consumes a maximal of 8×0.25 GPU cards. On multi-tenant clusters, the discrepancy between requested resource and real usage leads to the paradox of *long job waiting queue* and *low cluster resource usage*. This is inevitable since resource quota is occupied by workloads consuming much less GPU than claimed necessary.

Control stage and conditional execution offers a systematic approach to reconcile such discrepancy for DL workloads. Unlike conditional join, the control stage here can be materialized physically (usually into a singleton task). With DL binaries loaded, it can decide on proper distributed strategy and resource specification for each stage. Such decisions can be made by analyzing the operator-tree generated from TensorFlow compiler, in conjunction with resource usage profiling information collected from similar historical jobs. Alternatively, control task can also do a "dry-run" on sample input to gain insights on resource usage pattern. A possible conditional planspace can be given by $\mathcal{A}_P = \{\langle PS, Worker\rangle, \langle Worker\rangle\}$, in which `MultiWorkerMirroredStrategy` is represented by the $\langle Worker\rangle$ that describe scenarios when dedicated *PS* stage is deemed unnecessary, or when asynchronous training is unsuitable. Once the proper distributed strategy is chosen, control task also determines, and communicates to JM, the proper resource specifications. JM will then update the execution plan accordingly. With late binding on resource and distributed strategy, effective cluster-wide GPU utilization increases by over 35%, with *marginal performance degradation* and significant decrease in job queuing time.

4.3 Adaptive Modeling for Deep-Learning

In this sub-section, we share some observations and experiences from hosting Alibaba's core DL platform [26] on Fangorn. Particularly, we discuss how an adaptive execution framework offers the necessary flexibility to parallelize DL workloads effectively and efficiently, on shared clusters with heterogeneous hardwares.

Unlike relational workloads that readily translate into graphs of dozens, or even hundreds of stages, DL workloads are usually presented by much simpler execution graph. For example, ring-allreduce paradigm [37] is often presented as a one-stage graph, and parameter server [20] presented by two stages. The simplified representations can be largely attributed to lack of the need for *global* operations (e.g., aggregation or sort) in DL workflows. It allows complex DL operator-tree to be *independently* executed within a process, without stages chaining. The relatively simple graph have ushered attempts to parallelize DL workloads with map-only or map-reduce paradigms. Such attempts under-fit the DL workloads and ignore their unique data transportation and scheduling properties, as discussed in Section 3.3. Fangorn's rich semantics facilitates accurate modeling of DL executions by capturing their fundamental scheduling requirements, thereby enabling production-critical features such as job elasticity and customizable failover strategies. More importantly, it allows multi-mode DL workloads to be orchestrated in conformity of their unique properties. For example, DL training may require co-scheduling of all workers, and is usually iterative in nature. Large-scale batch DL inferences, on the other hand, finish in one epoch and each worker can be scheduled, reused, and released individually. The flexibility allows inference workloads leverage opportunistic scheduling with resources over-subscription [5, 40, 47]. Fangorn recognizes the distinct difference in scheduling various DL workloads, and decouples DL inference

workloads from co-scheduling strategies that used to govern all DL jobs. Substantial performance improvement is observed with this upgrade, especially on busy clusters: idle duration spent waiting for co-scheduling resources is reduced by orders of magnitude, while overall job latency improves by over 40%.

Additionally, computation in distributed DL workloads is mostly carried out by customized operators opaque to execution framework, which makes it hard to guarantee idempotence. This breaks execution "reentrancy" [14, 36], and hinders enabling fundamental capabilities such as elasticity or failover. In contrast to platform-and-algorithm-specific bare mental solutions [29, 33], Fangorn seeks to enable *interactive-adaptability* by involving DL engine in the decision makings, thereby achieving dynamic adjustment that can be applied generally. For example, the control node described in Section 4.2.2 introduces a secondary "master" in the graph. It can incorporate DL-specific logic to facilitate coherent adjustment of algorithmic parameters when graph structure, such as stage parallelism, is being modified. Coordinated global check-pointing among all workers is also made possible, to ensure that algorithmic convergence and correctness remain intact with dynamic reconfigurations.

Finally, the specialized hardware commonly used in DL workloads introduces additional dimension in scheduling decisions. For example, different placement strategies on NVLink [31] enabled GPUs can result in dramatic performance difference for DL workloads. Additionally, the iterative DL computation produces predictable hardware usage patterns that can be exploited to enable fine-grained resource sharing among various workloads, leading to significant boost in hardware utilization. All of these challenges call for a joint design of DL runtime, execution orchestration, and cluster resource scheduler. Some of these joint efforts have been reported in [42], but this remains largely an ongoing work that continues to generate challenges for Fangorn.

5 HYBRID EXECUTION FRAMEWORK

Gang scheduling all tasks in a graph offers the potential benefits of processing data immediately upon its production. Yet the benefits are only fully attainable when computation can be efficiently pipelined. In practice, data barriers, such as the commonly-used sort-based operators, regularly present themselves in distributed workloads, and pipeline-friendly operations rarely span across entire graph for complex workloads. Unconditional gang scheduling can thus lead to inefficient resource usage. On the other hand, leveraging pre-launched and long-running containers can noticeably reduce scheduling and task-launching overhead, therefore it has been widely adopted [4, 15, 38] for execution acceleration. However, this may not always be an option, since there are usually practical limitations imposed on admissibility to long-running containers:

a) Resource Usage Characteristics: A computation task may not be suitable to be scheduled to a pre-launched container, due to the task's excessive CPU / Memory usages, or dependency on special hardware (e.g., RDMA or FPGA device).

b) Security: Stringent security requirements may demand VM-based isolation strategy for certain types of computation, such as tasks containing UDF. The launching and disposing of VM can introduce impact that affects availability of long-running containers.

c) Runtime Duration: The benefits of leveraging pre-launched resources are especially noticeable for agile tasks that complete in shorter duration. Such benefits diminish for tasks taking longer to execute, making them unsuitable candidates to be admitted.

The ramifications of a task's inadmissibility into pre-launched container pool may differ with scheduling capabilities offered by the underlying framework. For systems that require entire job to use the same type of resources, *gang scheduled or not*, inadmissibility of one single task will reject the entire graph. For example, a seemingly straightforward relaxation from gang-scheduling is bubble execution illustrated in Fig. 6(c), and a similar framework was studied in [45]. However, bubble execution shares the same restriction as gang scheduling in terms of unanimous admissibility of all tasks. Its *exclusive* binding with one singular resource type hinders wide adoption in production. Firstly, one single task unsuitable for executing in long-running containers would disqualify entire graph from bubble execution. Secondly, it is generally unpractical to reduce bubble boundary inside a logical stage. The entailed requirement to rerun a massively-parallel stage on single task failure is usually unacceptable in production. Finally, for graphs containing stage(s) that can deplete entire container pool before its parallelism is satisfied, bubble execution is not an option either.

As profiled in Section 2, the majority of production data is processed by jobs of medium to large scales. It is therefore of paramount importance to optimize workloads with scales that fall into this range. However, gang-scheduling or bubble-scheduling these jobs over pre-launched containers is unrealistic with practical concerns such as failure cost, security, resource efficiency, or simply availability of pre-launched containers. In comparison, by leveraging the unique architecture of Fangorn, hybrid execution allows incorporation of various resource types and flexible scheduling strategies in one execution graph. This allows regions of a large graph, especially those on critical execution path, to benefit from expedited executions facilitated by "local" co-scheduling and data pipelining, even though the entire graph may not be all suitable for one universal execution strategy. Hence the exploration of job acceleration *becomes possible for jobs of all scales* with hybrid execution.

For hybrid execution, the graph shall be segmented so that sub-graph(s) suitable for co-scheduling can be identified. To this end, denote a graph as $\mathcal{G} = (S, E)$, where S is the set of stages and E set of edges. Parallelism of stage $s \in S$ is denoted by its cardinality $|s|$. e^S and e^D denotes source and destination of an edge $e \in E$. A graph $\mathcal{G}$ can be decomposed into a set of disjoint sub-graphs $\mathcal{G}_i$, s.t. $\mathcal{G}_i = (S_i, E_i)$, where $\bigcup_i S_i = S$ and $S_i \cap S_j = \emptyset, \forall i \neq j$. For *co-scheduled* sub-graphs specially denoted by $\vec{\mathcal{G}}_i = \left(S_i, \vec{E}_i\right)$, each edge $e \in \vec{E}_i$ has the property of $e^S, e^D \in S_i$, and can be annotated with physical properties {Transient, $*$, Source-Scheduled}. Here $*$ suggests the irrelevance of Shuffle-Pattern entry. In contrast, physical properties {Persisted, $*$, Source-Completed} apply to any edge $e \in E^C = E \backslash \left(\bigcup_i \vec{E}_i\right)$. For briefness of discussion, we refer to *co-scheduled sub-graph* simply as *sub-graph* hereon. A sub-graph segmentation algorithm iterates through $\mathcal{G}$ to identify all co-scheduled $\vec{\mathcal{G}}_i$s and is depicted in Algorithm 1.

Overall, acceleration offered by co-scheduling (sub)graphs can be attributed to two factors:*a)* removal of task launching overhead, including resource queuing, binaries download/loading, task initialization, and *b)* expedition offered by pipelining between stages.

Conversely, "imperfect" data pipeling introduces side-effects of resource inefficiency with co-scheduling. Algorithm 1 reflects a heuristic two-phased segmentation strategies. It allows Fangorn to work together with optimizer in a loosely-coupled way, in determining final sub-graph segmentation. Particularly, the input from optimizer is encapsulated in hints H, which may contain information regarding whether a barrier exists on output edge, or the complexity of operations hosted by a stage. For example, existence of sort-based operator in a stage is considered to *create output barrier*. In this regard, honoring the data barrier hint reduces possibility of idle-spinning in downstream tasks. Although such decision may *not* be most aggressive towards execution acceleration, it results in plans that promote more efficient resource usage.

Fangorn JM acts as the final arbitrator of graph segmentation, with empirical observations from workloads characteristics taken into account when determinating implementation details. For example, ascending or descending sorting on stages during initialization of Algorithm 1 corresponds to bottom-up *vs* top-down segmentation. *Caeteris paribus*, bottom-up approach is usually chosen since analytical workloads oftentimes translate into inverted-triangle-shaped graphs, therefore, it tends to include more pipeling-eligible stages. In addition, Fangorn incorporates information that may not be available during cost-based optimization, before arriving at final sub-graph segmentation. It makes judicious tradeoff decisions among various performance metrics based on real-time cluster statistics and workload SLAs. For instance, JM may allow a job with stringent latency-SLA to be segmented with partial data barrier *within* co-scheduling subgraphs, to tradeoff resource-efficiency for execution latency. Impact of different strategies is evaluated in Section 6.

Algorithm 1: Sub-graph Segmentation Algorithm

input : $\mathcal{G} = (S, E)$
output : segmented $\mathcal{G}$, with each of $\vec{\mathcal{G}}_i = \left(S_i, \vec{E}_i\right)$ identified
param : max sub-graph size C_{max}, hints set H
initialization: $E^c \leftarrow \emptyset$; $i, j \leftarrow 0$; $C_0 = 0$
$S_{sort} \leftarrow$ `sort`(S), by distance to root
while *not all stages visited* **do**
  $s_j \leftarrow S_{sort}[j++]$
  if CanAddToSubgraph(s_j, S_i) **then**
    do BFS starting from s_j **:** for connecting stage s', add to S_i until CanAddToSubgraph(s', S_i) returns `false`
    $i \leftarrow i + 1$
  else
    do: $\forall e$ with $e_D = s_j$, add e into E^c
  end
end

function CanAddToSubgraph(s, S_i)
 if $|s| + |S_i| > C_{max}$ ***or*** $\exists e \in E^C$, *s.t.* $e^D \in S_i$ ***or*** *any hints in* H *implies task(s) in* s *unsuitable for* S_i **then**
  return `false`
 else
  return `true`
 end
end function

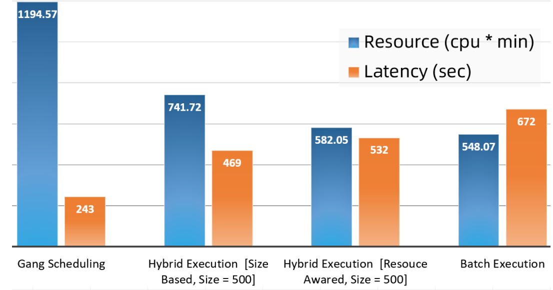

Figure 10: TPCH 1TB executed in various modes

6 PERFORMANCE EVALUATION

6.1 Performance with Resource Awareness

Evaluations of parallel data processing oftentimes focus on *latency* as the performance metric, since it reflects important system capabilities to finish computation as soon as possible. Real-world workloads, however, are subject to resource constraints. In fact, resource efficiency can sometimes be more critical than latency, since they usually translates to monetary cost. This is particular true for enterprise workloads at large scales.

Fangorn's flexible graph modeling facilitates versatile execution strategies to attain different tradeoff-points in the joint space of resource-efficiency and execution latency. In Fig. 10 we evaluate both metrics together for different execution modes against TPCH benchmark at 1TB scale. The evaluation is setup on a dedicated cluster with 15 machines, each having 96 cores, 256Gb RAM and 100T hard-drive. In addition to gang scheduling and batch scheduling, the hybrid executions are evaluated with two subgraph segmentation strategies. The first naïve *size-based* strategy with $H = \emptyset$, which favors more "aggressive" subgraph segmentation that aims to include as many tasks in the subgraph, as long as total task count does not exceed allowable size C_{max}. The second is a resource-aware strategy that takes into account the "effectiveness" of data-pipeling within segmented subgraphs, so that stages with barrier operator(s) may not be included in a subgraph, with hints H constructed to assist this decision. Both hybrid execution setups are bound by a max sub-graph size of $C_{max} = 500$.

As revealed by Fig. 10, with enough resource quota, gang scheduling entire graph erases overhead associated with tasks launching and allows pipeline acceleration *whenever possible*. However, the aggressive latency optimization is achieved at a significant premium on resource usage, since many query plans in TPCH are represented by *deep* graphs. Additionally, at 1TB scale, sort-based operators enacting pipeline barriers are commonly presented in execution plans. Both would increase probabilities of idle-spinning and are unfavorable for efficient resource usage. Meanwhile, batch execution consumes the least resource on the same workloads, but over a much longer duration. The two strategies adopted by hybrid executions deliver different tradeoff points between the two extremes. In particular, the resource-aware hybrid executions only co-schedule sub-graphs when doing so attains both benefits of efficient data pipeling, and reduction of overhead in task launching. Therefore, such strategy achieves notable performance improvement with marginal cost in resource consumption, when comparing to batch execution. The naïve size-based strategy, on the other hand,

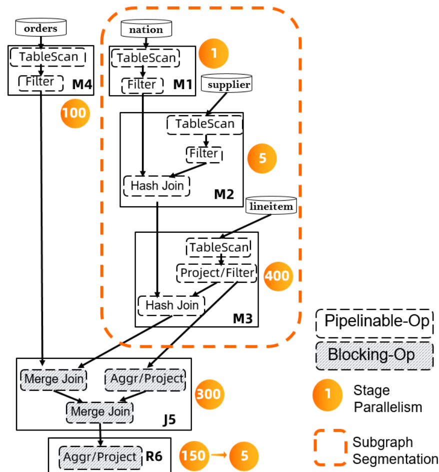

Figure 11: Q21 Execution plan and subgraph segmentation

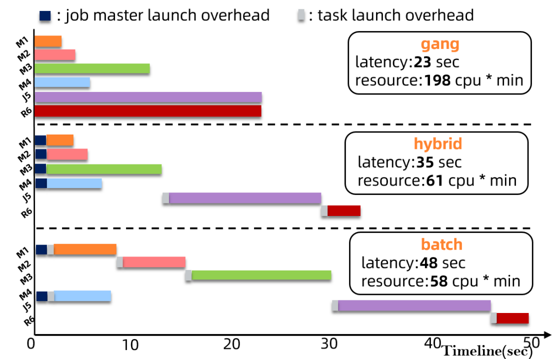

Figure 12: Q21 execution-timeline and resource usage

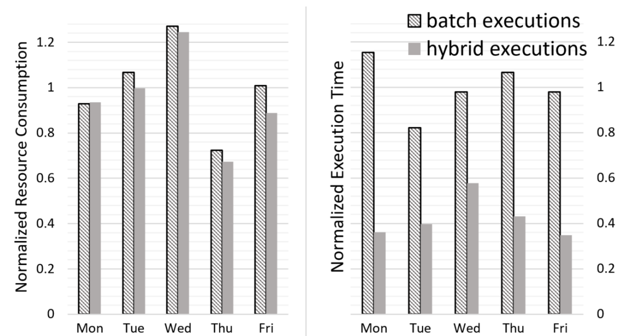

Figure 13: Impact of Hybrid-Executions in production

produces segmented subgraphs larger in both number and size, possibly with blocking barriers inside subgraphs. Consequently, it is not surprising that performance improvement is achieved with noticeably more resource usage.

To gain more insight on the impact of different execution strategies, we evaluate below the execution patterns of a typical query, namely TPCH-Q21. Particularly, we show how resource consumption and job latency differ with batch, gang, and *resource-aware* hybrid executions. Fig. 11 illustrates the query plan, with a total of 6 stages. The plan is generated with hints on which operators are blocking, to allow Graph Builder to annotate *pipelinable* edges. With that Fangorn JM groups $M1, M2, M3$ as a subgraph according to Algorithm 1. On the other hand, Fig.12 illustrates execution timelines in different modes. Several observations can be made:
a) Overhead of launching JM or spontaneously-requested task measures $1 \sim 2$s or less, which can be avoided with MGM and prelauched containers. Although the launch overhead may seem insubstantial, it can be notable for interactive scenarios. Additionally, the task launching overhead also accumulates across multiple stages.
b) Effective pipelining between stages can significantly accelerate execution, which can be noted in the execution of *M1, M2* and *M3* in different modes. Even root stage *M1* finishes much faster when it is co-scheduled with *M2* with data pipeling: partially due to the avoidance of data persisting phase at the end of computation.
c) Substantial amount of idling is introduced when gang scheduling stages that cannot be efficiently-pipelined, especially those located *deeper in the graph*. For example, execution of *R6* (and *J5*) span the entire job duration(23s) in gang execution mode. It only runs for a fraction of that duration in batch mode ($\sim$ 3s). Obviously, in this case the majority of time is spent idle waiting for upstream inputs, which leads to considerable resource waste. Gang scheduling also prohibits possibility of dynamic parallelism adjustment, which takes effect on *R6* for both hybrid and batch executions.

In addition to standard benchmarking, hybrid execution is now fully enabled on all production clusters at Alibaba, with resource-aware subgraph segmentation as the default execution strategy. Every day about *2.5 million* distributed jobs are executed with at least one subgraph, switching from the canonical batch executions. Fig. 13 shows the impact of this upgrade, in terms of both latency and resource usage. For fair comparison, queries were chosen from production work-flows repeatedly executed every day, with relative input size fluctuation $< 10\%$. As we can see, no notable increase in resource consumption is observed after the upgrade, while remarkably performance improvement is achieved. Actually, given the reduction in execution latency, overall resource consumption (as measured by cpu $\times$ time) declines after switching to hybrid executions. Evidently, the chosen subgraph segmentation strategy has produced plans favoring local gangs in which data can be efficiently pipelined and resource effectively leveraged. It can also be noted that latency improvement with hybrid execution on production clusters is more notable than benchmarking results (such as those studied in Fig. 10). This can be contributed to the distinct characteristics of resource availability and plans in production workloads. In many cases, hybrid execution avoid excessive latency of waiting for small amount of resource in resource-hungry clusters, which is different from controlled benchmarking. In addition, production workloads overall benefit more from hybrid execution, since large fraction of them contains pipelinable data-flow.

6.2 Scalability and Adaptability

With the amount of data processed daily measured by *Exabytes*, the ability to scale and accommodate various workloads is a fundamental requirement for the underlying execution framework to upholds the parallel data processing at Alibaba. Fangorn's scalability and versatility are recognized and attested by its full deployment across

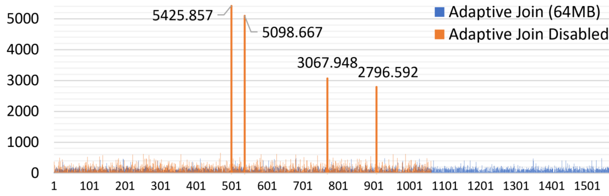

Figure 14: Time distribution among join tasks, TPCx-BB Q19

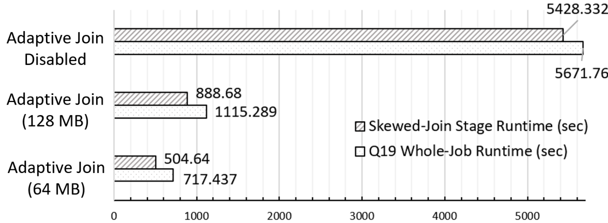

Figure 15: Runtime for TPCx-BB Q19 in different settings

the company, as well as being one of the few contestants to complete TPCx-BB suites *on one single execution framework*. As a newer benchmark suite, TPCx-BB measures comprehensive competency and performance of modern big-data systems, by incorporating various workloads including relational queries for structured data, and machine learning for semi-structured and unstructured data. At 30TB scale [35], Fangorn supports QPM (query per min) performance almost 3 times faster than runner-up submission, at less than $\frac{1}{3}$ the cost. In addition, it is *the only system* that scales to 100TB [34] as of July 2021. Evidently, the outstanding performance at such massive scale cannot be achieved without joint efforts from execution framework and the highly-optimized engines built-atop Fangorn. Both 100-TB and GB-scale workloads are executed on one unified framework, as Fangorn adapts to the scale, input volume, computation complexities of different workloads, choosing appropriate execution strategies automatically. We refer readers to official reports [34, 35] for complete evaluation setups.

Here we evaluate, as an example, how adaptability built within Fangorn helps significantly improve performance of TPCx-BB Q19, a query with data skewness that echoes with challenges observed on production workloads. Q19 explores relationship between high return and negative reviews, by joining `reviews` and `items` tables. The filtered `reviews` data fed into join, however, is notably skewed on several partitions, creating join skewness discussed in Sec. 4.1. The skewness is further amplified with compute-intensive UDF applied against join outcome that performs semantic analysis on review entries. On 100TB category, the time distribution on tasks execution, with and without Adaptive Join, is illustrated in Fig. 14. For the latter, several most-skewed tasks taking more than 90 mins to complete (as labeled), while most other tasks finished in less than 10 mins. In comparison, when Adaptive Join is enabled, with skew threshold set to be default value of 64*MB*, the skew is *automatically* detected and handled by balancing out the skewed partition, which accelerate the join stage by *more than 10 folds*. In addition, Fig. 15 provides insight on impact of adaptive join on overall job latency, with split thresholds of 64*MB* and 128*MB*. Although Q19 is executed on Fangorn as a graph of more than 10 stages, the skewed join dominates overall job latency, and significant improvement is achieved once that is addressed by adaptive join handling. We find

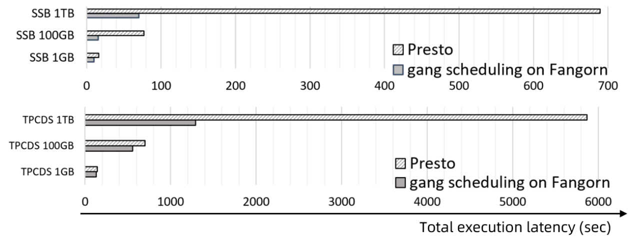

Figure 16: SSB and TPCDS results at various scales

this representative of production workloads as well, since joins are typically compute-intensive and time consuming in production too.

Finally, we compare performance of Fangorn-empowered relational engine on smaller-to-medium size dataset, against Star Schema Benchmark [23] and TPC-DS [25], at various scales. Fig. 16 shows the overall latency of executing both benchmarks with Fangorn, against Presto (v 0.224) on the same cluster. With latency being the evaluation metric, gang scheduling is chosen on Fangorn, and it compares favorably at all data-scales, especially when input size scales up to TB. It should be noted that performance on both systems are evaluated with default configurations at all scales, with no modification or user interventions in-between. Apparently, the end results is highly contingent on the implementation specifics of the relational engines, and both can potentially be fine-tuned for (possibly significant) further improvement. For example, using different settings and plugins at different scales is known to yield optimized results on Presto [38]. However, we present this evaluation result with two major takeaways: *Firstly*, as a general execution framework, Fangorn is not only capable of supporting efficient executions for massively-parallel workloads, but can also facilitate *interactive* queries. Remarkably, the *sub-second* average latency for SSB/TPCDS queries at 1GB scale, compares Fangorn favorably with those tailored for the state-of-art MPP engines, even at GB scales. *Secondly*, the adaptability of Fangorn, backed by various resources and versatile execution strategies, allows it to accommodate workloads of various scales, automatically.

7 CONCLUSION AND FUTURE WORKS

In this paper, we present Fangorn, the new generation of execution framework at Alibaba that orchestrates, every day, tens of millions of distributed jobs with multiple computation engines. With a descriptive graph model, Fangorn adapts to characteristics of various workloads and copes with uncertainties commonly presented on production clusters, by dynamically adjusting both physical and logical graphs during executions. The flexibility to leverage resources of different kinds in one single job allows Fangorn to explore a new class of hybrid-executions and battle-test them on production workloads. The adaptability and hybrid capabilities offered by Fangorn allow tradeoffs to be explored between performance and resource usage for heterogeneous workloads of all scales, and resolve practical challenges on production multi-tenant clusters.

Many dynamic adjustments discussed herein rely on Fangorn as an execution framework to adapt *automatically*. While it allows optimal transformations in many cases, the unilateral decision by Fangorn is not without its limits. Interactions between execution framework and computational engines are currently being actively explored, to allow cooperative adaptability that involves coherent adjustments among various components in the distributed system.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI' 16)*. 265–283.

[2] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.

[3] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. 2010. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*. 119–130.

[4] Eric Boutin, Paul Brett, Xiaoyu Chen, Jaliya Ekanayake, Tao Guan, Anna Korsun, Zhicheng Yin, Nan Zhang, and Jingren Zhou. 2015. JetScope: Reliable and interactive analytics at cloud scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1680–1691.

[5] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 14)*. 285–300.

[6] Nicolas Bruno, Sapna Jain, and Jingren Zhou. 2013. Continuous cloud-scale query optimization and processing. *Proceedings of the VLDB Endowment* 6, 11 (2013), 961–972.

[7] Nicolas Bruno, YongChul Kwon, and Ming-Chuan Wu. 2014. Advanced join strategies for large-scale distributed computation. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1484–1495.

[8] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1265–1276.

[9] Qi Chen, Cheng Liu, and Zhen Xiao. 2013. Improving MapReduce performance using smart speculative execution strategy. *IEEE Trans. Comput.* 63, 4 (2013), 954–967.

[10] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[11] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2014. Major technical advancements in Apache Hive. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1235–1246.

[12] Yuzhen Huang, Yingjie Shi, Zheng Zhong, Yihui Feng, James Cheng, Jiwei Li, Haochuan Fan, Chao Li, Tao Guan, and Jingren Zhou. 2019. Yugong: Geo-Distributed data and job placement at scale. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2155–2169.

[13] Shadi Ibrahim, Hai Jin, Lu Lu, Song Wu, Bingsheng He, and Li Qi. 2010. Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. IEEE, 17–24.

[14] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 59–72.

[15] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, et al. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Cidr*, Vol. 1. 9.

[16] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2010. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM symposium on Cloud computing*. 75–86.

[17] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2012. Skew-tune: mitigating skew in MapReduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 25–36.

[18] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[19] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal* 27, 5 (2018), 643–668.

[20] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 14)*. 583–598.

[21] Wei Li, Dengfeng Gao, and Richard Thomas Snodgrass. 2002. Skew handling techniques in sort-merge join. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 169–180.

[22] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 18)*. 561–577.

[23] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 237–252.

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc.

[25] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. 2002. TPC-DS, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 582–587.

[26] Alibaba Machine Learning Platform for AI. Accessed July 2021. https://www.alibabacloud.com/product/machine-learning.

[27] Alibaba MaxCompute. Accessed July 2021. https://www.alibabacloud.com/product/maxcompute.

[28] Distributed Strategy for Training on TensorFlow. Accessed July 2021. https://www.tensorflow.org/guide/distributed_training/.

[29] ElasticDL. Accessed July 2021. https://elasticdl.github.io/.

[30] Kubernetes. Accessed July 2021. https://kubernetes.io/.

[31] NVLink. Accessed July 2021. https://www.nvidia.com/en-us/data-center/nvlink/.

[32] Spark 3.0. Accessed July 2021. https://spark.apache.org/releases/spark-release-3-0-0.html.

[33] TorchElastic. Accessed July 2021. https://github.com/pytorch/elastic/.

[34] TPCx-BB 100TB Benchmark. Accessed July 2021. Official report, based on computation platform build on Fangorn. http://www.tpc.org/tpcx-bb/results/tpcxbb_result_detail5.asp?id=120100202.

[35] TPCx-BB 30TB Benchmark. Accessed July 2021. Official report, based on computation platform build on Fangorn. http://www.tpc.org/tpcx-bb/results/tpcxbb_result_detail5.asp?id=120100201.

[36] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. 2015. Apache TEZ: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*. 1357–1369.

[37] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[38] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.

[39] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[40] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache Hadoop Yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 1–16.

[41] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 20)*. 449–462.

[42] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 20)*. USENIX Association, 533–548. https://www.usenix.org/conference/osdi20/presentation/xiao

[43] Yu Xu and Pekka Kostamaa. 2009. Efficient outer join data skew handling in parallel DBMS. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1390–1396.

[44] Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. 2008. Handling data skew in parallel joins in shared-nothing systems. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 1043–1052.

[45] Zhicheng Yint, Jin Sun, Ming Li, Jaliya Ekanayake, Haibo Lin, Marc Friedman, José A Blakeley, Clemens Szyperski, and Nikhil R Devanur. 2018. Bubble execution: resource-aware reliable analytics at cloud scale. *Proceedings of the VLDB Endowment* 11, 7 (2018), 746–758.

[46] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments.. In *Osdi*, Vol. 8. 7.

[47] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. 2014. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1393–1404.

[48] Jingren Zhou, Per-Ake Larson, and Ronnie Chaiken. 2010. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 1060–1071.

[49] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3165–3166.