Swift: Reliable and Low-Latency Data Processing at Cloud Scale

Bo Wang*, Zhenyu Hou*, Yangyu Tao*, Yifeng Lu*, Chao Li*, Tao Guan*, Xiaowei Jiang*, Jinlei Jiang†
*Alibaba Group
†Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
*{yanyu.wb, zhenyuhou.hzy, yangyu.tyy, yifeng.lyf, li.chao, tony.guan, xiaowei.jxw}@alibaba-inc.com
†jjlei@tsinghua.edu.cn

***Abstract*—Nowadays, it is a rapidly rising demand yet challenging issue to run large-scale applications on shared infrastructures such as data centers and clouds with low execution latency and high resource utilization. This paper reports our experience with Swift, a system capable of efficiently running real-time and interactive data processing jobs at cloud scale. Taking directed acyclic graph (DAG) as the job model, Swift achieves the design goal by three new mechanisms: 1) *fine-grained scheduling* that can efficiently partition a job into $graphlets$ (i.e., sub-graphs) based on new shuffle heuristics and that does scheduling in the unit of graphlet, thus avoiding resource fragmentation and waste, 2) *adaptive memory-based in-network shuffling* that reduces IO overhead and data transfer time by doing shuffle in memory and allowing jobs to select the most efficient way to fulfill shuffling, and 3) *lightweight fault tolerance and recovery* that only prolong the whole job execution time slightly with the help of timely failure detection and fine-grained failure recovery. Experimental results show that Swift can achieve an average speedup of 2.11× on TPC-H, and 14.18× on Terasort when compared with Spark. Swift has been deployed in production, supporting as many as 140,000 executors and processing millions of jobs per day. Experiments with production traces show that Swift outperforms JetScope and Bubble Execution by 2.44× and 1.23× respectively.**

I. Introduction

A. Motivation

It is a lasting effort in the past decades to facilitate efficient processing of terabytes or even petabytes of data. As a result, many data-parallel computing models and frameworks have been put forward, including MapReduce [13], Dryad [27], Lusail [3], S4 [36], Storm [40], Hero [30], Spark [51], Pregel [32], PowerGraph [21], ProbeSim [31], Lux [29], Petuum [45], Mxnet [9], TensorFlow [2], Ray [35], Fractal [17], G-thinker [46], to name but just a few. Nowadays new changes are happening, bringing additional challenges to large-scale data processing. First, computing is increasingly done on shared infrastructures such as data centers and clouds. Second, the demand for real-time and interactive data analytics is surging as a response to the tough economic climate — reduced "time-to-insight" usually means competitive advantage in the market.

Running large-scale applications on shared infrastructures like clouds is challenging for the following reasons. First, application scheduling is in general an NP-hard problem with no polynomial solution. It is hard to develop a low-overhead approximate solution at cloud scale even using simple strategies, for many factors (e.g, the needed resources types, resource scale, the amount of applications under consideration) are involved and the scheduling goals (e.g., maximizing resource utilization and minimizing response time) are conflicting. For example, MaxCompute [10], a fully cloud-hosted data warehouse solution and service developed and run by Alibaba to support data warehouse and business intelligence (BI) analysis, web log analysis, transaction analysis of e-commerce sites, customer behavior analysis, and so on, involves not only terabytes (TB) or petabytes (PB) level data storage, but also computation and network (for data dissemination) across data centers. Each day, millions of jobs are processed, with the job running time ranging from seconds to hours or even days [47]. Second, workload usually changes from time to time. Moreover, various workloads on shared computing infrastructures may interfere with each other, making things even worse. Third, failures become common as computing scale increases. Handling failures introduces extra overhead and hurts performance.

Existing work to deal with the challenges can be roughly divided into two categories, namely resource-centered one and task-centered one. Resource-centered systems such as Mesos [25], Hadoop YARN [41], Fuxi [56], Apollo [8], Borg [42], and Hydra [12] work at the *resources management* level, trying to improve resource utilization by reallocating idle resources across applications. Task-centered systems such as Qincy [28], DRF [18], Sparrow [38], Paragon [14], Quasar [15], Tarcil [16], Firmament [20], Graphene [23], CARBYNE [22] and YARN-H/Tez-H [55] work at the *job scheduling* level, trying to improve application performance by effective and efficient scheduling (e.g., reducing interference between co-located workloads), accelerating the decision process, and incremental scheduling and problem-specific optimizations.

Though the above systems have improved resource utilization and ensured better application performance, but they are far from perfect to meet the need of real-time or interactive processing. For example, the overhead of job scheduling is still high as pointed out in [38], and the task launching overhead could not be omitted. In addition, due to the tough economy, it is imperative nowadays that applications should run with limited resources. Under such a condition, new systems are badly needed. Indeed, systems such as JetScope [7], Impala [33] and Bubble Execution [48] have been developed recently

2375-026X/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00272

to support low-latency interactive queries at scale. Though these systems reduce application latency, resource utilization is not as high as expected for two reasons. First, *resource fragmentation* is inevitable as resources are allocated, freed and re-allocated from time to time. Second, *resource waste* exists for the job consisting of multi-stages [27] because the allocated executors will keep idle until the required data are received from the predecessor tasks.

A trivial way to avoid resource waste and resource fragmentation is dividing the query execution graph into some sub-graphs [48] and scheduling each one respectively. However, this method still does not touch the root cause of worker idle-running, that is, worker is launched a long time before input data arrive. What's more, the mechanism to dump intermediate data between sub-graphs to disks and re-run the whole sub-graph when failure occurs is not performance friendly and recovery efficient.

B. Contribution

To deal with the above issues, this paper presents Swift, a reliable and low-latency data processing system at cloud scale. Swift has been deployed in production at Alibaba as a long-running service on tens of thousands of machines and hundreds of thousands of executors, processing more than four million jobs per day for years. Swift works around in-network shuffling. It achieves both low latency and high resource utilization by optimizing and synthesizing the well-known and best practice techniques in a systematic way, including the directed acyclic graph (DAG) job model, classical controller-worker architecture, fine-grained scheduling on the basis of shuffle-mode-aware job partitioning, and heartbeat-based failure handling. Our contributions with Swift are as follows.

Fine-grained scheduling with new job partitioning heuristics. Unlike JetScope [7] and Impala [33] that treat a whole job as the basic unit for scheduling and failure recovery, Swift presents new shuffle heuristics for logically partitioning a job DAG into fine-grained *graphlets* and does scheduling in the unit of a graphlet (Section III-A), thus avoiding resource fragmentation and waste.

Adaptive in-network data shuffling. In favor of the idea that data shuffling matters to performance especially for data-intensive applications, Swift defines several best practice memory-based in-network data shuffling schemes and presents a mechanism for the system to select the best one dynamically at runtime (Section III-B). Therefore, the costly inter-task data transfer time is effectively shortened.

Lightweight fault tolerance and recovery. Bearing in mind that failures come with cost and may harm job performance significantly, Swift improves heartbeat-based detection with machine health monitoring (Section IV-A), making failure detection timely and low overhead. In addition, it provides fine-grained failure recovery on the graphlet basis (Section IV-B), making failure handling effective and efficient.

The rest of this paper is organized as follows. Section II gives an overview of Swift. Section III details how Swift achieves low latency through fine-grained scheduling and adaptive in-network shuffling. Section IV explains the fine-grained failure recovery mechanism. Section V shows the experimental results. Section VI reviews the related work and the paper ends in Section VII with some conclusions.

II. An Overview of Swift

A. Job Description

Since SQL (Structured Query Language) is pervasive, Swift provides a SQL-like programming language as JetScope [7] does for users to describe their jobs in a familiar and efficient way. Fig. 1 shows how TCP-H Q9 is described in Swift language. Such a job is then converted to the DAG job model as shown in Fig. 4(a) by a parser or compiler program, which may exert some optimizations meanwhile. Since the full details of Swift programming language and the way to parsing it are out of the scope of this paper, we will stop here. In the following, we will assume that users submit jobs directly in DAG for the sake of simplicity. One thing to be pointed out here is that Swift supports all typical SQL operators such as $sort\ merge\ join$, $sort\ aggregate$, $window$, $order\ by$, and so on.

```
select nation, o_year, sum(amount) as sum_profit
from (
 select n_name as nation, substr(o_orderdate, 1, 4) as o_year,
      l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
 from tpch_supplier s
  join tpch_lineitem l on s.s_suppkey = l.l_suppkey
  join tpch_partsupp ps on ps.ps_suppkey = l.l_suppkey and ps.ps_partkey = l.l_partkey
  join tpch_part p on p.p_partkey = l.l_partkey
  join tpch_orders o on o.o_orderkey = l.l_orderkey
  join tpch_nation n on s.s_nationkey = n.n_nationkey
 where p_name like '%green%'
)
group by nation, o_year
order by nation, o_year desc
limit 999999;
```

Fig. 1. TPC-H Q9 described in Swift programming language. After parsing, it is converted to the DAG model shown in Fig. 4(a).

B. Architecture

Swift adopts the classical controller-worker architecture as JetScope [7] does to avoid two-round resource allocation of resource-centered systems [25], [41], [56], [42]. In this way, the scheduling latency is reduced. The detailed architecture is shown in Fig. 2, where Swift Admin acts as the controller and the shadow controller mechanism is enabled to avoid a single point of failure.

Swift Admin fulfills resource management and scheduling via *Executor Manager* and *Resource Scheduler*, job management and scheduling via *Job Scheduler* and *DAG Scheduler*. In order to reduce job scheduling latency further, Swift Admin runs in an event-driven way, with *Event Processor* handling various status- and resource-related events. It is the duty of *DAG Scheduler* to acquire resources from the *Resource Scheduler* and schedule tasks to the available Swift *Executors*.

The worker machine provides computing resources for tasks in terms of Swift *Executors*, which are pre-launched when Swift starts. After launched, the status of Swift *Executor* is reported to Swift Admin and then different actions are taken by the *Executor Manager* — the status will be cached in

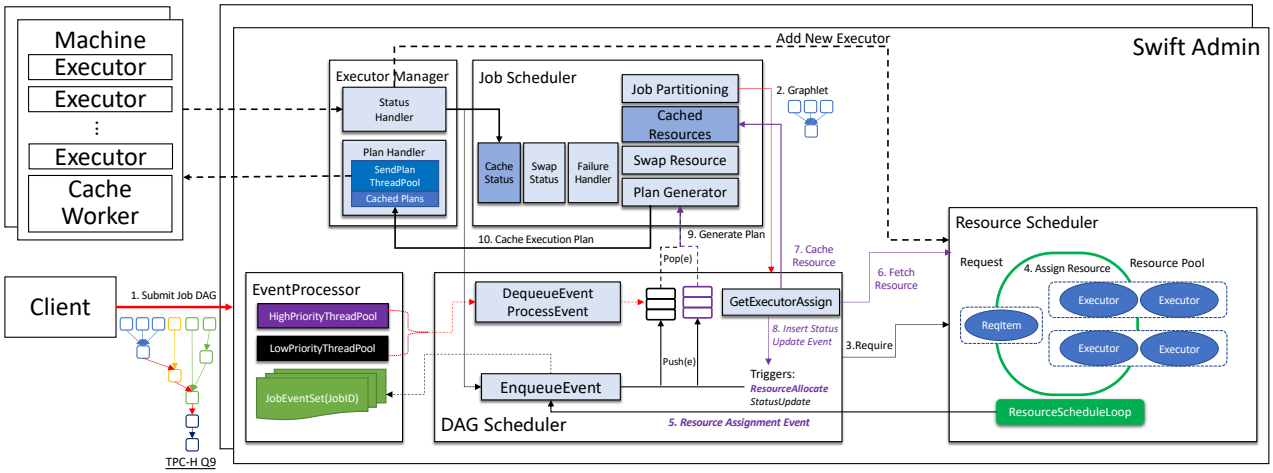


Fig. 2. The architecture of Swift. It follows the classical controller-worker architecture, with Swift Admin as the controller and Swift Executor as the worker. Swift works in an event-driven manner. To avoid a single point of failure, shadow controller mechanism is also supported as most distributed systems do today. Usually, there are dozens or hundreds of Swift Executors running on each machine in the real world.

the *Cache Status* of the *Job Scheduler* if the executor has a task to run and otherwise, it will be added to the resource pool maintained by the *Resource Scheduler*. On each worker machine, there is also one Swift *Cache Worker*, which is introduced to solve the TCP incast problem [54] associated with large jobs and will be detailed in Section III-B.

C. System Workflow

The workflow of Swift can be illustrated from the perspective of a job lifecycle. In Swift, a job lifecycle starts when the *Client* submits a job. We suppose the job is admitted. Otherwise, the job lifecycle ends. For each job admitted, the *Job Scheduler* will create a Job Monitor for the following job scheduling and status maintenance. The first step taken by the *Job Scheduler* after creating the Job Monitor is to analyze the dependency between stages and partition the job DAG into *graphlets* according to the inter-stage shuffle mode (Section III-A). Afterwards, the obtained *graphlets* are handed to the *DAG Scheduler*, which then registers resource requirements of *graphlets* with the *Resource Scheduler*. Each resource requirement is recorded as a request item (i.e., ReqItem in the figure) by the *Resource Scheduler*.

For each resource request received, the *Resource Scheduler* allocates all or part of the required resources according to the property of the *graphlet* as well as data locality and machine load. For more details, please refer to Section III-A. After resources are assigned, the related information is cached into the *Job Scheduler*, which then generates execution plan for each task accordingly. All plans are cached in the *Plan Handler* of *Executor Manager* and sent to certain Swift *Executors* for execution by a dedicated thread pool. When the execution completes, Swift Admin is notified. Afterwards, the *Executor Manager* records the status, and notifies the *DAG Scheduler* and the *Job Scheduler* for further scheduling or job status update.

The process keeps running in an event-driven way mediated by the *Event Processor* until the job completes. To keep the scheduling latency low, the *Event Processor* handles resource assignment events in high priority and fulfills event handling in a multi-threading way. When failures occur during the job execution, the *Failure Handler* of *Job Scheduler* determines the tasks to re-run according to the failure type and the fine-grained failure recovery strategy (Section IV). After that, the *DAG Scheduler* is notified to register resource requirement of these tasks to re-run with the *Resource Scheduler*. Finally, the tasks are re-executed.

III. Low Latency Optimizations

Swift achieves the goal of low latency and high performance through fine-grained scheduling and adaptive in-network shuffling. Details are as follows.

A. Fine-Grained Scheduling Based on Job Partitioning

Gang Scheduling, which schedules all tasks of a job at the same time to achieve higher concurrency, has been adopted by many interactive systems, such as JetScope [7] and Impala [33] for the low scheduling latency it can achieve. However, gang scheduling is not resource efficient for two reasons.

First, no task will be scheduled until all resources required by the job are allocated, resulting in *resource fragmentation*. Resource fragmentation means waste, for the idle resources cannot be put into use for running the task.

Second, for jobs consisting of multiple stages [27], there are dependencies among different stages. A typical example is the MapReduce job [13], where the Reduce task cannot be executed until all Map tasks finish sorting data and the data shuffle procedure completes. For these jobs, after tasks are gang scheduled to the executors, the resources will keep idle until the required data are received from the predecessor tasks, which obviously results in resources waste.

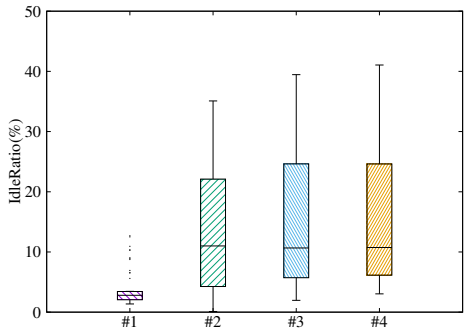


Fig. 3. The *IdleRatio* of 4 product clusters when gang scheduling is adopted. Obviously, gang scheduling results in resources waste.

To measure the degree of resources waste, we introduce a new metric called $IdleRatio$, which is defined as $IdleRatio = \frac{T_{data\_arrive} - T_{task\_start}}{T_{task\_finish} - T_{task\_start}}$, where T_{task_start} is the time when the task plan arrives at the executor, T_{data_arrive} is the time when the input data are ready, and T_{task_finish} is the task complete time. We measure the *IdleRatio* of jobs (the character of which is described in Section V-B) on 4 production clusters with gang scheduling, with the result shown in Fig. 3. Each cluster consists of over 10,000 machines, and the result shown is the average value got via the widely-used four quartile method [26]. We can see from the figure that the average $IdleRatio$ of 4 clusters is 3.81%, 13.15%, 14.45%, and 14.92% respectively. Obviously, a large quantity of resources have been wasted in gang scheduling.

To deal with the problem and improve resource utilization, Swift presents a fine-grained scheduling strategy that partitions the whole job DAG into several sub-graphs (called *graphlet* here) according to our new heuristics about inter-stage shuffling and does scheduling in the unit of *graphlet*.

1) Job Partitioning with New Shuffle Heuristics: Shuffle is common in the real world. For example, 97 out of 100 TPC-DS queries include at least 1 operation such as "order by", "group by" or "join", which may incur shuffle operation between tasks within a job. According to [53], over 50% of Spark jobs executed daily at Facebook have at least 1 shuffle operation. Shuffle matters to performance due to all-to-all transfer of growing size of data. It is in this sense that Swift works around shuffle for job partitioning and failure recovery.

To partition a job, we examine the shuffle mode of each edge in the job DAG. If it involves some global **SORT** operations (e.g., $StreamedAggregate$, $MergeJoin$, $Window$, $SortBy$, and $MergeSort$), data produced by the previous stages cannot be streamlined to the successor stage for continuous processing due to the operation semantics. The edge in such a case is called a *barrier* edge. Otherwise, it is called a *pipeline* edge. *Barrier* edges provide new *heuristics* for job partitioning. The detailed partitioning algorithm is shown in Algorithm 1 and Algorithm 2. It starts with one starting stage in the job DAG (Algorithm 1) and traverses the whole job DAG recursively (Algorithm 2) until the job DAG becomes empty. For each given stage, it is first added to the given sub-graph (i.e., *graphlet*) and then both its incoming and outgoing edges are checked. For stages associated with the *pipeline* edges, be they the successors or the predecessors, they will be removed from the job DAG and added to the same *graphlet*. Compared with the job partitioning scheme of Bubble Execution [48], our scheme is more straightforward and of lower complexity.

Algorithm 1 Shuffle-Mode-Aware Job Partitioning

```
Input:  Job_DAG;
Output: Graphlet_List;
while Job_DAG not empty do
    Get and remove the first stage from Job_DAG in topology order
    Construct a new graphlet
    Call scanAndAddStages(Job_DAG, stage, graphlet)
    Add graphlet to Graphlet_List
end while
```

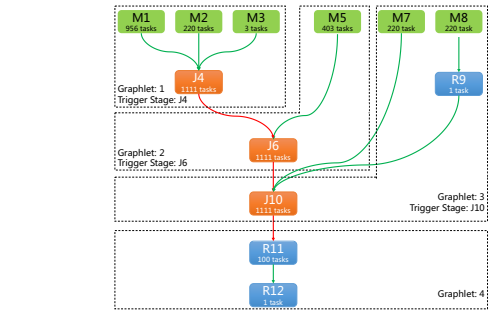
Algorithm 2 scanAndAddStages

```
Input: Job_DAG, stage, graphlet;
Add stage to graphlet
for each output of stage do
    if (output stage in Job_DAG) and (output edge is pipeline) then
        Remove output stage from Job_DAG
        Call scanAndAddStages(Job_DAG, output stage, graphlet)
    end if
end for
for each input of stage do
    if (input stage in Job_DAG) and (input edge is pipeline) then
        Remove input stage from Job_DAG
        Call scanAndAddStages(Job_DAG, input stage, graphlet)
    end if
end for
```

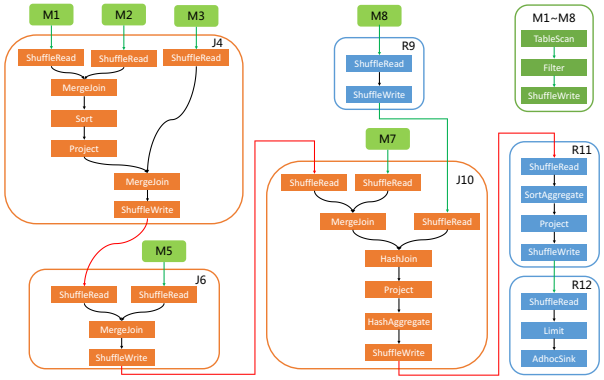
Fig. 4(a) illustrates how the job DAG of TPC-H Q9 is partitioned, where the barrier and pipeline edges are drawn in red and green respectively. As shown in Fig. 4(b), J4, J6, and J10 contain $MergeSort$ operator, thus the edges between J4 and J6, J6 and R10, J10 and R11 are barrier edges, and the whole Q9 DAG is partitioned into 4 graphlets: graphlet 1 consisting of M1, M2, M3, and J4; graphlet 2 consisting of M5 and J6; graphlet 3 consisting of M7, M8, R9, and J10; and graphlet 4 consisting of R11 and R12.

2) Fine-Grained Scheduling of Graphlets: After a job DAG is partitioned into graphlets, the *DAG Scheduler* submits them one by one to the *Resource Scheduler*. When assigning resources, both data locality and machine load are considered. Data locality is considered to accelerate data loading, which is time-consuming especially for data-intensive applications [49], [52], [24], [43]. Machine load is considered to avoid scheduling flock, that is, too many tasks are scheduled to a small number of machines in favor of data locality while the other machines are idle, which obviously leads to poor performance. For tasks without locality preference, the most free machine is chosen. For each graphlet received, gang scheduling is used to ensure low latency and high concurrency. Since graphlet is of finer granularity than the whole job DAG, the problem of *resource fragmentation* is alleviated at least to some extent.

The submission order of graphlets is determined by the dependencies among them. A graphlet can be submitted only when all its input data are ready. Take the TPC-H Q9 DAG



(a) Job Partitioning



(b) Operators of Each Stage

Fig. 4. An example of job partitioning with TPC-H Q9, where 4 graphlets are obtained. The operations of each stage are also shown.

shown in Fig. 4(a) as an example. Graphlet 1 is submitted first, and graphlet 2 can be submitted only after J4 finishes, and then comes graphlet 3 after J6 completes. Graphlet 4 is submitted last after J10 completes. Please note such an order is somewhat conservative to graphlet 3, for tasks M7 and M8 can be executed concurrently with graphlet 2. We do so to avoid the resources waste caused by J10 waiting for the input data from J6.

B. Adaptive Memory-based In-Network Shuffling

Data shuffling matters to performance, for the involved operations such as TCP connections establishment and data transfer are all time-consuming. For a network shuffle of M predecessor tasks and N successor tasks, at least $M \times N$ TCP connections should be established. To reduce the overhead of establishing so many TCP connections and alleviate the TCP incast problem [54], file-based shuffling is put forward. A typical example can be found in Dryad [27]. Nevertheless, file-based shuffling is still time-consuming and can be improved [53]. In Swift, we present an adaptive memory-based in-network shuffling scheme to optimize performance. As illustrated in Fig. 5, we define three types of shuffle, namely **Direct Shuffle**, **Local Shuffle**, and **Remote Shuffle** for jobs to select dynamically at runtime.

Direct Shuffle sends shuffle data directly from the predecessor tasks to the successor tasks. It has the advantage of the least memory copy times. However, it suffers from the overhead of too many ($M \times N$) TCP connections establishment, the TCP incast problem [54], and a lot of TCP retransmissions when large quantity of tasks are involved.

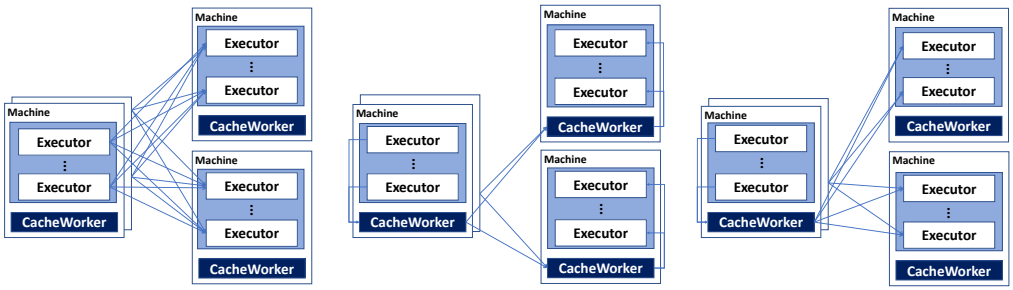
Local Shuffle directly writes/reads shuffle data to/from the *Cache Worker* on the local machine. Data shuffle is fulfilled by the *Cache Workers*. As mentioned in Section III-A, there are two types of edges between stages, namely the *barrier* edge, and the *pipeline* edge. Different steps are taken for them. For the *pipeline* edge, both writer and reader tasks are gang scheduled. The *Cache Worker* on the writer side sends shuffle data to the destination *Cache Worker* as soon as they are ready. After the destination *Cache Worker* receives the desired shuffle data, the reader tasks are notified to do further processing. As for the *barrier* edge, since the writer task and the reader task belong to different graphlets, it is most likely the reader task has not been scheduled when the writer task is finished. Therefore, the shuffle data is only written to the writer *Cache Worker*, and the destination *Cache Worker* will pull the data proactively after the reader task is put into running.

Suppose there are M predecessor and N successor tasks running on Y machines, the number of TCP connections needed between Swift *Executors* and the *Cache Workers* is $M + N + C_Y^2$ at most, where $C_Y^2 = \frac{Y!}{2!(Y-2)!}$. Since each machine can run tens of *Executors*, Y is much smaller than M and N. Thus, Local Shuffle has the least TCP connections between tasks. But, compared with Direct Shuffle, it introduces two additional times of memory copy.

Remote Shuffle writes shuffle data to the *Cache Worker of* local machine and the successor tasks pull data directly from the *Cache Worker*. The number of TCP connections needed is at most $M+N \times Y$, which is less than $M \times N$ of Direct Shuffle and greater than $M+N+C_Y^2$ of Local Shuffle. Compared with Direct Shuffle and Local Shuffle, Remote Shuffle has modest memory copy times.

To achieve the best performance, Swift selects the right shuffle type adaptively according to the shuffle size, namely the number of edges between all source stage tasks and the sink ones. Direct Shuffle is used for small-sized shuffle, Local Shuffle for huge-sized shuffle, and Remote Shuffle for middle-sized shuffle. In our production settings, the threshold of shuffle size is set to 10,000 and 90,000 respectively.

Memory Management of the Cache Worker Local Shuffle and Remote Shuffle write shuffle data to the *Cache Workers* and delete them to release memory after they have been consumed by all successor tasks. Since most jobs in Swift are short and small (refer to Section V-B for more details), shuffle data only exists in the *Cache Worker* for a short time and the space could be reused by other jobs in a timely fashion. In the case that memory shortage does happen, which is only of the probability less than 1% in our production clusters, LRU (Least Recently Used) algorithm [37] is used to swap *old* data to disks. Since this can be done in large data chunk, it would not hurt performance greatly.



(a) Direct Shuffle (b) Local Shuffle (c) Remote Shuffle

Fig. 5. Three types of data shuffling schemes defined by Swift. Multiple *Cache Workers* are used in (b) and (c) to provide enough memory space.

IV. Lightweight Failure Handling

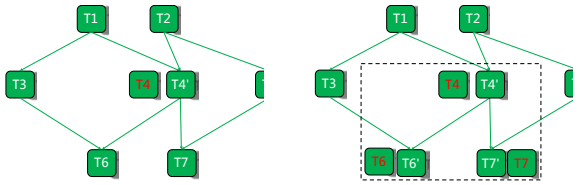
Failures are common in large scale distributed systems. Providing favorable fault tolerance and efficient failure recovery with low overhead is crucial for low latency data processing system. The most straightforward way to handle failures is to re-run the whole job. Given that failures could occur at any time on any machine, this way would waste a lot of resources — for the already complete tasks should be re-executed — and make the job execution time unnecessary long. Therefore, fine-grained and low-overhead fault tolerance and recovery mechanisms are badly needed. Swift achieves the purpose on the basis of job partition. Below are the details.

A. Timely Failure Detection

Detecting failures timely is a prerequisite for fast failure recovery. However, it is really challenging because numerous factors are involved in large scale distributed systems. For example, network congestion, machine crashes, and hardware exceptions usually do not lead to application level failures immediately. But when such failures are detected, harm to performance has been made. To deal with the problem, Swift presents the following three lightweight mechanisms.

First, Swift Admin keeps tracking all the *Executor* processes in a lazy and passive way — it is up to the *Executor* itself to report its status once the state changes. When an *Executor* process is launched, it reports its PID (process ID), TCP port and so on to Swift Admin. Once the process is re-launched due to some failures, its status is also reported to Swift Admin. In this way, Swift Admin could know process restart and initiate the failure handling process immediately in low overhead.

Second, Swift maintains periodic heartbeats between Admin and *Executors*. The heartbeat interval is vital in large scale systems. A larger interval means a longer delay in failure detection, which may cause greater harm. A shorter interval, on the other hand, means a heavier burden of Swift Admin and may harm system scalability. As a compromise, two strategies are exploited by Swift: i) a heartbeat manager is deployed on each machine as a proxy of all *Executors* on that machine to communicate with Swift Admin. In this way, the burden of Swift Admin is eased, and ii) heartbeat interval is carefully selected according to the cluster scale (5s, 10s, 15s for small, medium, large cluster respectively) so that no great harm can be done before the failure is detected.



(a) idempotent task case (b) non-idempotent task case

Fig. 6. Two cases of failure recovery within a graphlet.

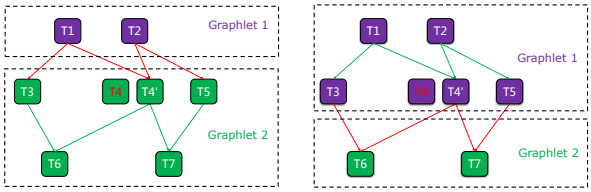
Third, Swift Admin also monitors the health of each machine. Once a machine failure is detected, all *Executors* on that machine are revoked by Swift Admin and if there are tasks running on those *Executors*, a failure recovery process will be triggered immediately. When a machine is found unhealthy (e.g., a large quantity of tasks on the machine failed in a short time), Swift Admin will mark it as **read-only** and stop scheduling new tasks to it. *Executors* on read-only machines will keep running until no more task is left unfinished in them. Then, the resources are revoked.

B. Fine-Grained Failure Recovery

Swift leverages a fine-grained failure recovery mechanism that only re-runs the failed tasks to minimize the impact of failure handling on job performance as much as possible. The mechanism works on the basis of graphlet described in Section III-A. According to the position where a failure occurs, three cases are distinguished as follows.

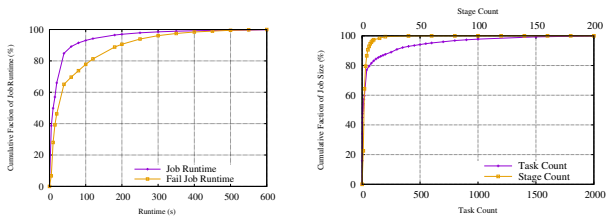
1) Intra-Graphlet Failure Recovery: In this case, the failed task, its predecessors and successors are all within the same graphlet. Fig. 6(a) shows such an example, where the failed task is T4, its predecessors are T1 and T2, and its successors are T6 and T7. According to the scheduling mechanism described in Section III-A, these tasks are gang scheduled. In the real production, there are two types of task, namely the idempotent one and the non-idempotent one. Corresponding to them, different steps are taken for failure recovery.

a) Idempotent Task Recovery: For idempotent tasks, their output data sets as well as the sequence of data within a set remain unchanged no matter how many times they run. For



(a) input failure case (b) output failure case

Fig. 7. Two cases of failure recovery across graphlets.



(a) Job runtime distribution (b) Job size distribution

Fig. 8. The characteristics of jobs in the production traces.

these tasks, the data sent to the successors could still be used even if they fail. Therefore, the recovery steps are simple. Take the failure of T4 in Fig. 6(a) as an example and suppose T4 has been executed. If T6 and T7 have received the desired data from T4, no step will be taken. Otherwise, a new instance T4' is initialized to replace T4. Meanwhile, T1 and T2 are notified to update their output channels to T4' and re-send the shuffle data to T4' without re-running. Afterward, T4' is run to regenerate the data required by T6 and T7.

b) Non-Idempotent Task Recovery: Unlike idempotent tasks, both the output data set and the sequence of data within a set may be different when non-idempotent tasks run many times. When failure occurs with these tasks, the data could not be reused even if they have flowed into the successors of depth. To recover such a failure, we need to re-run the failed task as well as all its successors that have been executed. Fig. 6(b) shows such an example, where the failed task T4 and its successors T6 and T7 are re-launched as T4', T6', and T7'.

2) Input Failure Recovery: Input failure occurs across graphlets, where the failed task and its predecessors belong to different graphlets. Fig. 7(a) shows an example, where the failed task T4 and its predecessors T1 and T2 belong to two different graphlets. According to Section III-A, <T1, T4> and <T2, T4> are *barrier* edges, and T1 and T2 only write shuffle data to the local *Cache Worker* for T4 to fetch. When T4 is re-launched as T4', there is no need to notify T1 and T2 to update their output channels, for T4' will fetch the desired data directly from their *Cache Workers*. As for the output channels of T4', they will be handled following the routine of *Intra-Graphlet Failure Recovery*.

3) Output Failure Recovery: Output failure also occurs across graphlets, but the failed task and its successors belong to different graphlets. Fig. 7(b) shows an example, where the failed task T4 and its successors T6 and T7 belong to two different graphlets. For this case, the input channels are updated following the routine of *Intra-Graphlet Failure Recovery*, and no step is needed for the output channels because T4' only writes shuffle data to the local *Cache Worker*.

C. Avoiding Useless Failure Recovery

There are some other failures caused by the application logic, for example, memory access violation and access to non-existent files or tables. For these failures, re-running the corresponding tasks does not help, but wastes resources. Therefore, once such failures occur, we just report them to the *Job Monitor* and no further recovery step is taken in order to save resources.

V. Evaluation

Swift has been thoroughly evaluated against various workloads using different settings in terms of performance, resource utilization, fault tolerance, and scalability.

A. Platform and Settings

Two clusters running Red Hat 4.1.2-50 are used for Swift evaluation. One cluster has 100 nodes, each of which is equipped with two 16-core Xeon E5-2682 v4 CPUs (40MB Cache, 2.5GHz), 192GB DDR3 RAM, 12 7TB SATA hard disks. The other cluster has 2000 nodes, each of which has two 6-core Xeon E5-2630 CPUs (15MB Cache, 2.3GHz), 96GB DDR3 RAM, 13 2TB SATA hard disks. All the nodes of the two clusters are connected via a 10GB Ethernet NIC (Network Interface Controller).

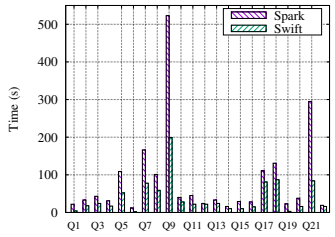
B. Workloads

The workloads used for Swift evaluation include TPC-H and some traces collected from production clusters with tens of thousands of machines. The traces involve 2000 jobs, with the job characteristics shown in Fig. 8(a) and Fig. 8(b). Obviously, most jobs are short and small: the average job run time is 30s, more than 90% of the jobs could complete in 120s, and more than 80% of the jobs consist of no more than 80 tasks and 4 stages. Each day, there are more than 4 million jobs running on Swift. To better illustrate the real performance of a production cluster, we also run some jobs as background workloads in all the evaluations.

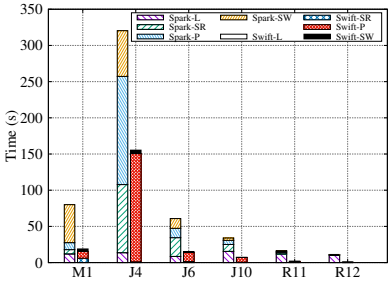
C. Execution Performance

We compare Swift with Spark, one of the most popular open-source systems, with the results discussed below.

1) Performance against TPC-H: We run TPC-H jobs with 1TB data to evaluate Swift performance and compare the result with that of Spark SQL 2.4.6. For fairness, Spark SQL is carefully tuned, with Parquet data format supported and cost-based table analyzing enabled. For Spark SQL, only the job running time is counted whereas the query analysis time is omitted. The cluster of 100 nodes is used for TPC-H evaluation, with the result shown in Fig. 9(a). For all the queries, Swift gets a total speedup of 2.11× over Spark.



(a) The result of TPC-H jobs with 1TB data



(b) Detailed result of the 4 phases of TPC-H Q9 job

Fig. 9. Performance comparison of Swift and Spark against TPC-H jobs, where L, SR, SW and P mean task launching, shuffle reading, shuffle writing, and record processing phase respectively. Particularly, SR for M1 is table scanning, and SW for R12 is adhoc sinking.

TABLE I
COMPARISON OF SWIFT AND SPARK WITH TERASORT JOBS

Job Size	Spark (s)	Swift (s)	Speedup
250×250	61	19	3.07
500×500	103	26	3.96
1000×1000	233	33	7.06
1500×1500	539	38	14.18

For both Spark and Swift jobs, each task execution could be subdivided into 4 phases, i.e., task launching that covers package downloading and executor launching, data shuffle writing, data shuffle reading, and record processing. We further investigate the performance of Spark and Swift on Q9 and show the detailed 4-phase execution time of critical tasks in Fig. 9(b). We can see that the benefit mainly comes from the following two mechanisms of Swift: 1) long running executors, which avoids the overhead of packages downloading and spares executors launching time — for Spark, launching all the critical tasks takes over 71s, and 2) adaptive in-network shuffling, which takes 8.92s for shuffle read and 9.61s for shuffle write, while saving and loading shuffle data to/from disks in Spark take 137.8s and 133.9s, respectively.

2) Comparison of Swift and Spark with Terasort Jobs: Table I shows the result, where the cluster of 100 nodes is used, the job size $M \times N$ (M, N = 250, 500, 1000, 1500) means the job has M Terasort Map tasks and N Terasort Reduce tasks, and each Terasort Map task processes 200MB data. The job execution time on Spark increases as the job size increases. When the job size is larger than 1000×1000, the job execution time shoots up. In contrast, the job execution time on Swift only increases slightly as the job size increases.

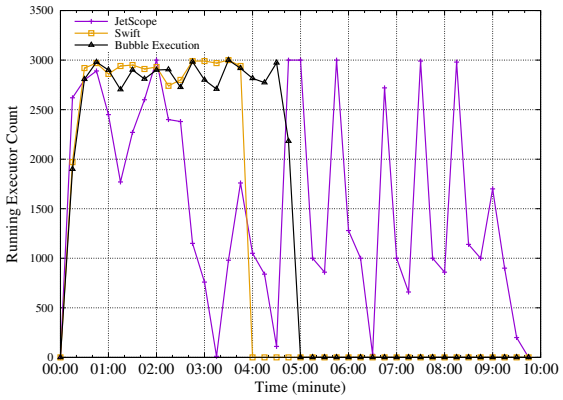


Fig. 10. Statistics of the number of running Executors when JetScope, Bubble Execution, and Swift are deployed on the 100-nodes cluster respectively.

Fig. 11. Normalized execution latency of 2,000 jobs with JetScope, Bubble Execution, and Swift on the 100-nodes cluster.

The larger the job size, the greater the speedup that can be achieved. The result also implies that Swift has better efficiency than Spark, for the same cluster is used.

D. Resource Utilization and Latency

We implement JetScope and Bubble Execution ourselves and compare them with Swift in terms of resource utilization and job latency by replaying the production job traces on the 100-nodes cluster. Fig. 10 and Fig. 11 show the results from different perspectives. Since Swift *Executor* is the basic unit to run tasks, the number of running Executors is used as an indicator of resource utilization.

Since jobs in JetScope are treated as a whole for scheduling, no job will be scheduled until there are enough free resources for it. Therefore, for a large job submitted, it will wait for a long time before all the required resources are available. Obviously, the CPU cycles of the free executors are wasted during the time. The fluctuation of the number of running Executors in Fig. 10 indicates bad resources utilization — the process is full of waiting and waste. As a result, as shown in Fig. 11, it is of the largest job latency — more than 60% of jobs are with a latency 2× greater than that of Swift.

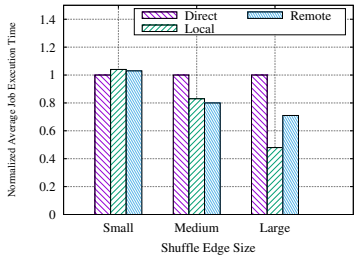
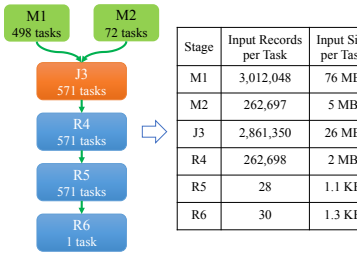


Fig. 12. The normalized average job execution time when Direct Shuffle, Local Shuffle, and Remote Shuffle are deployed respectively.



Stage	Input Records per Task	Input Si per Tas
M1	3,012,048	76 MI
M2	262,697	5 MB
J3	2,861,350	26 MI
R4	262,698	2 MB
R5	28	1.1 KI
R6	30	1.3 KI

Fig. 13. The details of the TPC-H Q13 job.

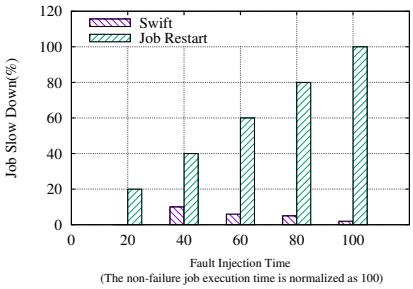


Fig. 14. The impact of failure recovery on a single job execution time between Swift and job restart policy.

As for Swift and Bubble Execution [48], since both of them can do scheduling in finer granularity (*graphlet* in Swift and *bubble* in Bubble Execution), free executors can be timely scheduled and put into use. Therefore, resources are highly utilized and resource fragmentation is slight. We can see from Fig. 10 that the number of running executors of Bubble Execution is almost the same as that of Swift. So does the job latency — as shown in Fig. 11, nearly 90% of jobs have a latency 1.5× less than that of Swift. As shown in Fig. 10, Swift and Bubble Execution can finish all jobs in 240s and 296s respectively, imposing a speedup of 2.44× and 1.98× over JetScope.

Swift performs better than Bubble Execution because: 1) Bubble Execution partitions a job DAG according to the shuffle data size and suffers from high partitioning overhead and long-time waiting — the assigned executors keep idle until the required data are ready, and 2) the adaptive in-network shuffling mechanism employed by Swift is more efficient than the disk-based shuffling mechanism of Bubble Execution.

E. The Benefit of Adaptive In-Network Shuffling

We evaluate the benefit of adaptive in-network shuffling by replaying a set of production job traces on the cluster of 2,000 nodes. The chosen jobs are divided into three categories according to the shuffle edge size, that is, *small-shuffle-sized*, *medium-shuffle-sized*, and *large-shuffle-sized*. The job traces are replayed three times with Direct Shuffle, Local Shuffle, and Remote Shuffle respectively. To better illustrate the result, for each category, the average job run time obtained with Direct Shuffle is normalized to 1.

As shown in Fig. 12, for *small-shuffle-sized* jobs, using Direct Shuffle can get the best average execution time, whereas using Local Shuffle and Remote Shuffle can slow down job execution by 4% and 3% respectively. For *medium-shuffle-sized* jobs, using Remote Shuffle can get the best average execution time, whereas using Direct Shuffle and Local Shuffle can slow down job execution by 25% and 3.8% respectively. For *large-shuffle-sized* jobs, using Local Shuffle can get the best average execution time, whereas using Direct Shuffle and Remote Shuffle can slow down job execution by 108.3% and 47.9% respectively. The reasons are given below.

For *small-shuffle-sized* jobs, only a small number of TCP connections are needed for data shuffling and the overhead of connection estabnlishment is negligible. Remote Shuffle and Local Shuffle slow down job execution mainly because of the additional memory copies introduced. As the shuffle edge size increases, more TCP connections are needed and the system spends more time in constructing connections. That's why Direct Shuffle performs poor. An investigation into the execution log shows that: 1) estabnlishing a TCP connection would take hundreds of milliseconds in a congested network, and for a task with hundreds of successors, it usually takes dozens of seconds to build all the TCP connections when Direct Shuffle is used; 2) TCP retransmission rate increases as the number of connections, and for *large-shuffle-sized* jobs, Direct Shuffle can incur a rate as high as 3%, whereas Local Shuffle and Remote Shuffle only incur a rate less than 0.02%.

F. Fault Tolerance

We use TPC-H Q13, which is detailed in Fig. 13, to evaluate the fault tolerance capability of Swift and compare it with job restart policy. We manually inject failures into different tasks and measure the failure recovery time and the impact of failure recovery on job execution time. The result is shown in Fig. 14, where the non-failure job execution time is used as a baseline (normalized to 100) and 5 failures are injected at time 20, 40, 60, 80, and 100 to M2, J3, R4, R5, and R6 respectively. At each run, only one failure is injected.

As shown in Fig. 14, there is no slowdown of the job execution time for Swift when the first failure occurs at time 20. The reason is that the failed task (M2 here) has sent its output to the following task (i.e., J3). However, the failure at time 40 causes a significant job slowdown. The reason behind

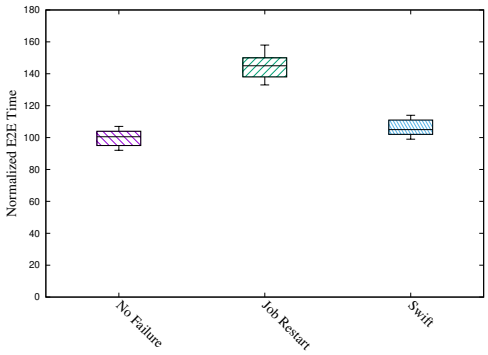


Fig. 15. The impact of failure recovery on performance with the real-world traces and failures.

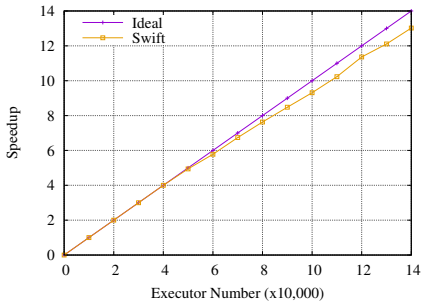


Fig. 16. The scalability of Swift (strong scaling).

is that the failed task J3 is on the critical job path and most importantly, of the large input data size. For all the failures, the fine-grained failure recovery mechanism of Swift only slows down the job by less than 10%, which is much lower than the case of job restart recovery.

Failures in the real world are diverse and more complicated [19], [6]. They can occur at any time during job execution. As shown in Fig. 8(a), about 50% failures occur within 30s and 90% within 200s. To show how Swift performs in dealing with the real-world failures, we redo the failure recovery experiment. First, we replay the traces without failure injection as the baseline and normalize the end-to-end job execution time to 100. Then, we replay the traces with failures regenerated according to the production traces and measure the end-to-end jobs execution time with job restart policy and with the fine-grained failure recovery mechanism shipped by Swift respectively. Fig. 15 shows the result, where the values are obtained with the widely-used four quartile method [26]. Failure recovery by job restart slows down job execution by 45% on average, whereas the fine-grained failure recovery mechanism of Swift only incurs an average slowdown of 5%.

G. Scalability

In this section, we measure the scalability of Swift with the cluster of 2,000 nodes. The workload is generated according to the production traces. We replay the same workload several times with different numbers of Swift *Executors*. The end-to-end run time with 10,000 Swift *Executors* is normalized as the baseline. Fig. 16 shows the result, where the ideal result is also shown. From the figure we can see that Swift can achieve near-linear scalability when the number of Executors increases from 10,000 to 140,000.

VI. Related Work

Many factors are involved for a distributed system like Swift. Below lists some most related work.

Resource utilization and QoS. At the resources level, Mesos [25], Hadoop YARN [41], Fuxi [56], and Borg [42] are developed and deployed to improve resource utilization by reallocating idle resources across applications while keeping QoS (Quality of Service). At the tasks level, Qincy [28], DRF [18], Sparrow [38], Paragon [14], Quasar [15], Tarcil [16], Firmament [20], Graphene [23], CARBYNE [22], YARN-H/Tez-H and so on are proposed to improve application performance by effective and efficient scheduling (e.g., reducing interference between co-located workloads), decision process acceleration, incremental scheduling and problem-specific optimizations.

In-memory computing. Many data processing systems such as Spark [51], MapReduce Online [11], Scuba [4], Dremel [34], and Shark [44] employ memory to manipulate various data so as to eliminate the IO bottleneck and shorten job execution. Swift also favors this idea to shuffle data in memory. Moreover, several in-network shuffle types are distinguished for adaptive selection based on shuffle edge size. In this way, the data transfer time between tasks is reduced.

Job scheduling. JetScope [7], Impala [33] and Bubble Execution [48] use *gang scheduling* to achieve higher concurrency. But as aforementioned, gang scheduling suffers from *resources fragmentation* because the whole job is treated as a basic unit in scheduling. Bubble Execution alleviates the problem by dividing a job graph into *buddles* (i.e., sub-graphs) and scheduling them respectively. However, resources waste still exists, for executors in Bubble Execution are launched long before the input data arrive.

DAG partitioning. Breaking a job DAG into stages and scheduling them independently have already been supported by systems like Spark [51] and Sparrow [38]. However, existing DAG partition metrics are simple and only work in specific scenarios. For example, there is no shuffle dependency for tasks of the same stage in Spark, but data dependency does exist between *graphlets* in Swift. Data shuffling in Spark is disk-based and of poor performance, whereas the adaptive in-memory shuffling scheme presented by Swift is resource-efficient and high-performance.

Failure detection. Timely failure detection is crucial in large-scale distributed systems. Heartbeat-based mechanisms have been widely used by many distributed frameworks [41], [56], [42], [7]. Swift takes a step further to introduce heartbeat manager, machine health monitor and the **read-only** mechanism to ease the detection burden and to save the executed cycles as much as possible.

Fault tolerance. F1 [39] uses coarse-grained fault tolerance which is simple but not efficient. Spark [51] achieves fault tolerance by leveraging RDD-based lineage re-computation [50]. JetScope [7] provides support for fined-grained fault tolerance via recomputing the failed tasks. MillWheel [5] and Flink [1] use checkpoint to achieve exactly-once fault tolerance. However, failures in the real world are more complicated and difficult to handle even with a small graph. Swift presents mechanisms covering both inter/intra-graphlet failures and idempotent/non-idempotent tasks. It can guarantee both performance and data validity.

VII. Conclusion

We have presented Swift, a reliable and low-latency large-scale data processing system in production. Working around shuffling, Swift adopts an event-driven controller-worker architecture to achieve both low latency and high resource utilization by leveraging fine-grained job scheduling on the basis of shuffle-aware job partitioning, adaptive memory-based in-network shuffling, and lightweight fine-grained failure recovery. The experimental results show that, compared with Spark, Swift can achieve an average speedup of 2.11× on TPC-H and 14.18× on Terasort jobs. Swift has been deployed in production on tens of thousands of machines, supporting as many as 140,000 executors and processing millions of jobs per day. Experiments with production traces show that Swift could achieve a speedup of 2.44× over JetScope and 1.23× over Bubble Execution.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments and suggestions for improving this paper. The work of Jinlei Jiang is sponsored by National Key R&D Program of China (2018YFB0204100). Corresponding Authors: Bo Wang (yanyu.wb@alibaba-inc.com) and Jinlei Jiang (jjlei@tsinghua.edu.cn).

References

[1] Apache flink: Stateful computations over data streams. https://flink.apache.org/.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI'16*, pages 265–283, 2016.

[3] Ibrahim Abdelaziz, Essam Mansour, Mourad Ouzzani, Ashraf Aboulnaga, and Panos Kalnis. Lusail: a system for querying linked data at scale. *Proceedings of the VLDB Endowment*, 11(4):485–498, 2017.

[4] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. Scuba: diving into data at facebook. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, 2013.

[5] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of the Vldb Endowment*, 6(11):1033–1044, 2013.

[6] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 51–68, Carlsbad, CA, October 2018. USENIX Association.

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

[8] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14*, pages 285–300, 2014.

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[10] Alibaba Cloud. Maxcompute: conduct large-scale data warehousing with maxcompute. https://www.alibabacloud.com/product/maxcompute.

[11] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Usenix Conference on Networked Systems Design and Implementation, NSDI'10*, pages 21–21, 2010.

[12] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 177–192, Boston, MA, February 2019. USENIX Association.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation, OSDI'04*, pages 137–149, 2004.

[14] Christina Delimitrou and Christos Kozyrakis. Paragon:qos-aware scheduling for heterogeneous datacenters. *Acm Sigarch Computer Architecture News*, 48(4):77–88, 2013.

[15] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *The 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'14*, pages 127–144, 2014.

[16] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the 6th ACM Symposium on Cloud Computing, SoCC'15*, pages 97–110, 2015.

[17] Vinícius Vitor dos Santos Dias, Carlos H. C. Teixeira, Dorgival O. Guedes, Wagner Meira Jr., and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1357–1374. ACM, 2019.

[18] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation, NSDI'11*, pages 323–336. USENIX, 2011.

[19] Phillipa Gill, Navendu Jain, and Nachi Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM Special Interest Group on Data Communications (SIGCOMM '11)*. ACM, August 2011.

[20] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16*, pages 99–115, Savannah, GA, November 2016. USENIX Association.

[21] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, pages 17–30, 2012.

[22] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16*, pages 65–80, Savannah, GA, November 2016. USENIX Association.

[23] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16*, pages 81–97. USENIX, 2016.

[24] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *2011 IEEE 27th International Conference on Data Engineering, ICDE'11*, pages 1199–1208. IEEE, 2011.

[25] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation, NSDI'11*, volume 11, pages 22–22, 2011.

[26] Rob J Hyndman and Yanan Fan. Sample quantiles in statistical packages. *The American Statistician*, 50(4):361–365, 1996.

[27] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review, SIGOPS'07*, 41(3):59–72, 2007.

[28] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy:fair scheduling for distributed computing clusters. In *IEEE International Conference on Recent Trends in Information Systems*, pages 261–276, 2009.

[29] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A distributed multi-gpu system for fast graph processing. *Proceedings of the VLDB Endowment*, 11(3):297–310, 2017.

[30] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD'15*, pages 239–250. ACM, 2015.

[31] Yu Liu, Bolong Zheng, Xiaodong He, Zhewei Wei, Xiaokui Xiao, Kai Zheng, and Jiaheng Lu. Probesim: scalable single-source and top-k simrank computations on dynamic graphs. *Proceedings of the VLDB Endowment*, 11(1):14–26, 2017.

[32] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, SIGMOD'10*, pages 135–146. ACM, 2010.

[33] Kornacker Marcel, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, and Casey Ching. Impala: A modern open-source sql engine for hadoop. In *7th Biennial Conference on Innovative Data Systems Research, CIDR'15*, 2015.

[34] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, et al. Dremel: a decade of interactive sql analysis at web scale. *Proceedings of the VLDB Endowment*, 13(12):3461–3472, 2020.

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

[36] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops, ICDMW'10*, pages 170–177. IEEE, 2010.

[37] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[38] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles, SOSP'13*, pages 69–84. ACM, 2013.

[39] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, et al. F1: A distributed sql database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013.

[40] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. Storm@twitter. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 147–156, 2014.

[41] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, and Siddharth Seth. Apache hadoop yarn:yet another resource negotiator. In *Proceedings of the 4th ACM Symposium on Cloud Computing, SoCC'13*, pages 1–16, 2013.

[42] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Tenth European Conference on Computer Systems, Eurosys'15*, page 18, 2015.

[43] Bo Wang, Jinlei Jiang, and Guangwen Yang. Actcap: Accelerating mapreduce on heterogeneous clusters with capability-aware data placement. In *2015 IEEE International Conference on Computer Communications, INFOCOM'15*. IEEE, 2015.

[44] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD'13*, pages 13–24. ACM, 2013.

[45] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

[46] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M. Tamer Ozsu, Wei-Shinn Ku, and John C. S. Lui. G-thinker: A distributed framework for mining subgraphs in a big graph. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1369–1380, Los Alamitos, CA, USA, apr 2020. IEEE Computer Society.

[47] Renyu Yang, Yang Zhang, Peter Garraghan, Yihui Feng, Jin Ouyang, Jie Xu, Zhuo Zhang, and Chao Li. Reliable computing service in massive-scale systems through rapid low-cost failover. *IEEE Transactions on Services Computing*, 10(6):969–983, 2017.

[48] Zhicheng Yin, Jin Sun, Ming Li, Jaliya Ekanayake, and Haibo Lin. Bubble execution: Resource-aware reliable analytics at cloud scale. *Proceedings of the VLDB Endowment*, 7(11):746–758, 2018.

[49] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems, EuroSys'10*, pages 265–278. ACM, 2010.

[50] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *The 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, pages 15–28, San Jose, CA, 2012. USENIX.

[51] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10*, pages 10–10. USENIX, 2010.

[52] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08*, page 7. USENIX, 2008.

[53] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. Riffle: Optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[54] Jiao Zhang, Fengyuan Ren, and Chuang Lin. Modeling and understanding tcp incast in data center networks. In *2011 IEEE International Conference on Computer Communications, INFOCOM'11*, pages 1377–1385, 2011.

[55] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. History-based harvesting spare cycles and storage in large-scale datacenters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16*, pages 755–770. USENIX, 2016.

[56] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: A fault-tolerant resource management and job scheduling system at internet scale. In *Proceedings of the VLDB Endowment*. VLDB Endowment, 2014.